CIDR 2011 January 9-12, 2011 Asilomar, California

Conference Proceedings

CIDR 2011 Conference Organization

Conference Chairs

- **Anastasia Ailamaki**, EPFL
- **Michael Franklin**, UC Berkeley
- **Joe Hellerstein**, UC Berkeley

Program Committee Chair

- **Michael Franklin**, UC Berkeley

Local Arrangements Chair

- **David DeWitt**, Microsoft and Wisconsin

CCC Liaison (OIV Track)

- **Hank Korth**, Lehigh University

Gong Show Organizer

- **Yanlei Diao**, UMass Amherst

Program Committee

- **Daniel Abadi**, Yale
- **Gustavo Alonso**, ETH Zurich
- **Sihem Amer Yahia**, Yahoo! Research
- **Magdalena Balazinska**, Washington
- **Michael Cafarella**, Michigan
- **Michael Carey**, UC Irvine
- **Ugur Cetintemel**, Brown
- **Yanlei Diao** , UMass Amherst
- **AnHai Doan**, Wisconsin and Kosmix
- **Johannes Gehrke**, Cornell
- **Stavros Harizopoulos**, HP
- **Meichun Hsu** , HP
- **Theodore Johnson**, AT&T
- **Hank Korth**, Lehigh University
- **Donald Kossmann**, ETH Zurich
- **Boon Thau Loo** , U Penn
- **Sam Madden**, MIT
- **Sergey Melnik**, Google
- **Christopher Olston**, Yahoo! Reasearch
- **Yannis Papakonstantinou**, UC San Deigo
- **Hamid Pirahesh**, IBM Almaden
- **Neoklis Polyzotis**, UC Santa Cruz
- **Sunita Sarawagi**, IIT Mumbai
- **Donovan Schneider**, Salesforce.com
- **Eugene Shekita**, IBM Almaden
- **Jun Yang**, Duke

CIDR 2011 Table of Contents

OUTRAGEOUS IDEAS AND VISION III: DEALING WITH PEOPLE

CLOUD SERVICES

ANALYTICS

Keynote (Tuesday January 11th, 8:30 am)

Data, Decisions, and Intelligence: Amidst the Foothills of a Revolution

Eric Horvitz, Microsoft Research

Systems that learn and reason from streams of data promise to provide extraordinary value to people and society. A confluence of advances has led to an inflection in our ability to collect, store, and harness large amounts of data for generating insights and guiding decision making. Given the long-term possibilities, we have been trekking through the foothills of a larger revolution--and have much to learn. Beyond the core goal of providing valuable services, fielding systems in the open world is critical for testing the sufficiency of existing algorithms and models, and often frames new directions for research. I will discuss efforts on learning and inference in the open world, highlighting key ideas in the context of projects in transportation, energy, and healthcare. Finally, I will discuss opportunities for creating systems with new kinds of competencies by weaving together multiple data sources and models.

Eric Horvitz is a Distinguished Scientist at Microsoft Research. His interests span theoretical and practical challenges with developing systems that perceive, learn, and reason. His contributions include advances in principles and applications of machine learning and inference, search and retrieval, human-computer interaction, bioinformatics, and e-commerce. He has been elected a Fellow of the Association for the Advancement of Artificial Intelligence (AAAI) and of the American Association for the Advancement of Science (AAAS). He currently serves on the NSF Computer & Information Science & Engineering (CISE) Advisory Board and on the council of the Computing Community Consortium (CCC). He received his PhD and MD degrees at Stanford University. More can be found at http://research.microsoft.com/~horvitz.

Flash Device Support for Database Management

Philippe Bonnet
IT University of Copenhagen
Rued Langaard Vej 7
Copenhagen, Denmark
phbo@itu.dk

Luc Bouganim
INRIA Paris-Rocquencourt & PRiSM/UVSQ
Domaine de Voluceau
Le Chesnay, France
Luc.Bouganim@inria.fr

ABSTRACT

While disks have offered a stable behavior for decades - thus guaranteeing the timelessness of many database design decisions, flash devices keep on mutating. Their behavior varies across models and across firmware updates for the same model. Many researchers have proposed to adapt database algorithms for existing flash devices; others have tried to capture the performance characteristics of flash devices. However, today, we neither have a reference DBMS design nor a performance model for flash devices: *database researchers are running after flash memory technology.* In this paper, we take the reverse approach and we define how flash devices should support database management. We advocate that flash devices should provide DBMS with more control over IO behavior without sacrificing correctness or robustness. We introduce the notion of *bimodal* flash devices that expose the full potential of the underlying flash chips as long as the submitted IOs respect a few well-defined constraints. We suggest two approaches for implementing bimodal flash devices: (a) based on the narrow block device interface, or (b) based on a rich interface that allows a DBMS to explicitly control IO behavior. We believe that these approaches are natural evolutions of the current generation of flash devices, whose complexity and opacity is ill-suited for database management. We discuss how bimodal flash devices would benefit many existing techniques proposed by the database research community, and identify a set of new research issues.

1. INTRODUCTION

For some time now, flash devices have been poised to replace disks as secondary storage [12]. Today, many different types of flash devices are finding their way into the memory hierarchy of database management systems (DBMS), from SSD to PCI-based racks (e.g., fusionIO and RamSan) and energy efficient FAWNs [5]. However, despite significant efforts [2, 9, 8, 17, 21, 29, 31, 20, 32], a reference design for database management with flash devices has yet to emerge[1].

Indeed, flash devices do not exhibit consistent characteristics. They embed a complex software called Flash Translation Layer (FTL) in order to hide flash chip constraints (erase-before-write, limited number of erase-write cycles, sequential page-writes within a flash block). A FTL provides address translation, wear leveling and strives to hide the impact of updates and random writes based on observed update frequencies, access patterns, temporal locality, etc. Their performance characteristics and energy profiles vary across devices [9, 8]. For instance, random writes are faster than reads on FusionIO's ioDrive [7] while random writes are much slower than the other operations on the Samsung model [9]. For some devices, performance varies in time based on the history of IOs, e.g., the performance of the Intel X25-M varies by an order of magnitude depending on whether the device is filled with random writes or not. What is the value of a DBMS design based on a storage subsystem whose behavior is not well understood and keeps on mutating?

By contrast, successive generations of disks have complied with two simple axioms: *(1) locality in the logical address space is preserved in the physical address space; (2) sequential access is much faster than random access.* As long as hard disks remained the sole medium for secondary storage, the block device interface proved to be a very robust abstraction that allowed the operating system to hide the complexity of IO management without sacrificing performance. The block device interface is a simple memory abstraction based on read and write primitives and a flat logical address space (i.e., an array of sectors). Since the advent of Unix [30], the stability of the interface and the stability of disks characteristics have guaranteed the timelessness of major database system design decisions, i.e., pages are the unit of IO with an identical representation of data on-disk and in-memory; random accesses are avoided (e.g., query processing algorithms) while sequential accesses are favored (e.g., extent-based allocation, clustering).

We must address the tension that exists between the design goals of flash devices and DBMS. Flash device designers, especially SSD and PCI-based racks designers, aim at hiding the constraints of flash chips to compete with hard disks providers. They also compete with each other, tweaking their FTL to improve overall performance, and masking their design decision to protect their advantage. Database systems, on the other hand, control the IOs they issue.

[1]We do not consider in this paper architectures providing direct access to the flash chips, e.g., embedded flash [4]

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

What database systems designers need is a clear and stable distinction between efficient and inefficient IO patterns, so that they can adapt their allocation strategies, data representation or query processing algorithms to the characteristics of the underlying storage devices. They might even be able to trade increased complexity for improved performance and stable behavior across devices.

So the problem is the following: How can flash devices provide DBMS with guarantees over IO behavior? Interestingly, flash chips already provide a clear, stable distinction between efficient patterns (page reads, sequential page-writes within a block), and inefficient patterns (in-place updates). Our key insight is that flash devices should expose this distinction instead of aggressively mitigating the impact of inefficient patterns at the expense of the efficient ones (e.g., trading reduced read performance to obtain improved random writes). In this paper, we introduce the notion of *bimodal* flash device that expose this dichotomy to the upper layers, thus providing efficient and predictable IO behavior for database systems. In terms of design, we see two approaches:

1. *Narrow Interface Device*: The most immediate approach is to keep the existing block device interface. Since flash devices have no knowledge of the manipulated data, we should (a) let the DBMS optimize its accesses and, (b) avoid uncontrolled FTL optimizations. This approach is similar, in spirit, to the way DBMSs interact with virtual memory [30]. Plagiarizing Stonebraker, this consists in replacing a "not quite right" service provided by the flash device with a comparable, application specific, service within the DBMS. The key questions here are: What abstractions should the FTL provide? How to handle the increased complexity at the DBMS level?

2. *Rich Interface Device* An alternative approach would be to rely on a rich interface to let DBMS and flash device collaborate on how to optimize performances. Indeed, there is an emerging consensus that the block interface is too narrow [28, 22, 23, 25, 26]. Coping with the block interface forces flash devices to perform complex tasks (i.e., wear leveling, garbage collection) independently from the application, possibly against its best interest. The key questions here are: What information should be passed from the DBMS to the flash device so that it can optimize its performance[2]? What kind of optimizations can be defined on flash devices? How should we design DBMSs to leverage a rich flash device interface?

We see Narrow and Rich bimodal devices as natural evolutions of the current generation of flash devices whose complexity and opacity is ill-suited for database management. Such an evolution is particularly important for database machines designers (e.g., Oracle's Exadata or Neteeza's TwinFin) that have to specify well-suited flash components for their systems. More generally, we understand that SSD manufacturers will only move if the gain is clear for their business. We see here an opportunity for the database community to influence the evolution of flash devices for the benefits of commodity database systems.

In this paper, we define the guarantees that a FTL should provide to a DBMS and derive the notion of bimodal flash device, we outline the design of Narrow and Rich bimodal flash devices and we explore how they will impact database management. Throughout the paper, we illustrate how ideas expressed in the literature would benefit from these new classes of devices. We also describe new research challenges.

2. FLASH DEVICES

At their core, flash devices rely on NAND flash chips that store data in independent arrays of memory cells. Each cell accommodates 1, 2, or 3 bits of information (SLC, MLC, TLC). Each array is a *flash block*, and rows of memory cells are *flash pages*[3].

The Good. A single flash chip can offer great performance (e.g., 40 MB/s Reads, 10 MB/s write) with low energy consumption [8]. Thus, tens of flash chips wired in parallel can deliver hundreds of thousands IOs per second. At the chip level, random operations are as fast as sequential ones. Recent flash chips can interleave operations and include multiple independent planes, thus processing operations concurrently [3]. A flash device is composed of a collection of flash chips, wired in parallel to a controller. The controller includes some cache (e.g. 16-32MB), potentially safe with respect to power failure [3]—e.g., cache can be RAM with capacitors or other NVM (eg. PCM). From this perspective, the potential of flash device is impressive.

The Bad. Unfortunately, flash chips have severe constraints: *(C1) Write granularity.* Writes must be performed at a page granularity[4]. *(C2) Erase before write.* A costly erase operation must be performed before overwriting a flash page. Even worse, erase operations are only performed at the granularity of a flash block (typically 64 flash pages). *(C3) Sequential writes within a block.* Writes must be performed sequentially within a flash block in order to minimize write errors resulting from the electrical side effects of writing a series of cells[5]. *(C4) Limited lifetime.* SLC, MLC and TLC flash chips can support respectively up to 10^6, 10^5, $5{\times}10^4$ erase operations per flash block. The trend is that flash chips store more bits per cell (e.g., TLC) with a smaller process geometry (e.g., 25 nm), larger page size, larger number of page by blocks and smaller lifetime. None of these evolutions challenge the nature of C1-C4.

And the FTL. The controller embeds the so-called Flash Translation Layer (FTL) in order to hide the aforementioned constraints. Typically, the FTL implements out-of-place updates to handle C2 using some reserved flash blocks called *log blocks*. Each update leaves, however, an obsolete flash page (that contains the before image). Over time such obsolete flash pages accumulate, and must be reclaimed by a *garbage collector*. A mapping between the logical address space exposed by the FTL and the physical flash space is necessary

[2]Note that we focus on IO performance; we do not consider integrating high-level database abstractions within storage devices, e.g., active disks [24]. Whether a rich interface naturally leads to active disks is a topic for future work.

[3]Flash pages may further be broken up into *flash sub-pages*

[4]While the write granularity is the page (4KB-8KB) for MLC NAND (no sub-pages), the read one can be smaller [31]. Actually, read granularity depends on the ECC sector size (512B - 1KB).

[5]Electric side effects may generate write errors effectively managed with error correction codes (ECC) at the hardware level.

to handle writes smaller than a flash page (C1), updates (C2), random writes (C3), and to support wear leveling techniques (C4), which distribute erase operations across flash blocks and mask bad blocks. This mapping is implemented based on a mapping table located in the controller cache, on flash or both [13]. *Page mapping*, with a mapping entry for each flash page generates large maps that do not fit in the controller cache for large capacity devices. *Block mapping* reduces drastically the mapping table to one entry per flash block [15]—the challenge is then to minimize the overhead for finding a page within a block. Thus, block mapping does not support random writes and updates efficiently. More recently, many *Hybrid mapping* techniques [19, 18, 13, 15] have been proposed that combine block mapping as a baseline and page mapping for log blocks. Besides mapping, existing FTL algorithms also differ on their wear leveling algorithms [10], as well as their log blocks management and garbage collection methods (block associative [16], fully associative [19], using detected patterns [18] or temporal locality [15]).

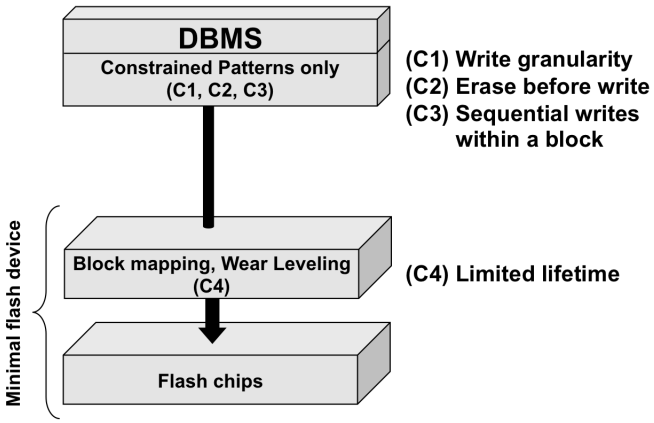


Figure 1: Minimal FTL does not support updates or random writes but offers optimal performance for IO patterns respecting C1, C2 and C3.

3. TUNNELING DBMS IO

Our overall problem is to resolve the tension that exists between FTL and DBMS design goals. From the point of view of a DBMS designer, the trivial solution is to take the FTL out of the equation and let a DBMS directly access flash chips. Obviously, the first step that DBMS designers would take is to introduce modularity to hide the complexity of dealing with flash chips, in effect re-introducing some form of FTL. So, the interesting question is not how to bypass the FTL, but what kind of FTL can DBMS designers rely on?

3.1 Minimal FTL

To start with, let us define the minimal level of service that a FTL should provide—because a DBMS cannot. The idea is that a minimal FTL would give maximal control and maximal stability to the DBMS. Let us look back at the constraints imposed by flash chips. First, a DBMS can trivially issue IOs at the granularity of a flash page (C1). This would require the FTL to advertise how flash pages should be aligned at the logical level. Second, a DBMS could rely on the *Trim* command[6] to tell a flash device to erase a flash block before it is written (C2). This would require the FTL to expose an abstraction of flash blocks at its interface. Third, a DBMS could write flash pages in sequence within flash blocks (C3). This would require the FTL to advertise how flash blocks are aligned, and how flash pages are mapped into flash blocks. Fourth, a DBMS cannot implement wear-leveling (C4). It must be provided by the FTL. Indeed, wear-leveling is absolutely necessary to guarantee the lifetime of a device. Flash device manufacturers will never release a product that can wear-out after some minutes of focused and intensive write/erase cycles.

One can thus imagine building flash devices with an FTL that implements wear leveling and provides abstractions for flash pages (a *logical page*) and flash blocks (a *logical block*). Such a FTL would rely on a block level map which is compact enough to be efficiently kept in the flash device safe cache (e.g., 16MB cache for 1 TB of 256 KB flash blocks)[7] With such a FTL, the DBMS would have to handle constraints C1-C3 —i.e., the DBMS could not submit IOs for in-place updates or random writes. On the other hand, the FTL would guarantee that the IOs that respect these constraints would be tunneled to the underlying flash chips as directly as possible (see Figure 1).

Would flash device manufacturers be interested in providing such FTLs? Probably not. Could DBMS designer always circumvent random writes on flash devices? Maybe in some cases (e.g. if flash devices are only used for immutable data sets), but definitely not for general-purpose databases.

3.2 Bimodal FTLs

To resolve the tension between FTL and DBMS design goals, we propose a bimodal FTL which achieves optimal performance as long as the DBMS manages constraints C1-C3, while providing best effort performance for all other IO patterns[8]. Interference between these two modes of operations should be minimized (see Figure 2).

While flash device designers have focused on efficiently enforcing the flash chip constraints for updates or random writes, there has not been much work on characterizing optimal performance for a flash device. More precisely, what is the most optimal mapping that a FTL can achieve? We have seen that a FTL must at least implement a form of block mapping to support wear-leveling. A mapping is thus optimal if (a) the block look-up is performed in the controller cache, and (b) the offset of the page within the block is derived from the logical address (i.e., consecutive logical addresses are written sequentially within a block). In the following, a flash block for which mapping is optimal is called an *optimal block*.

A bimodal FTL must provide optimal mapping for those

[6]The Trim command has been introduced in the ATA interface standard [28] to communicate to a flash device that a range of LBAs are no longer used by an application

[7]Otherwise, the block mapping has to be partially written on flash (as was the case in early block mapping FTLs [11]).

[8]A bimodal FTL must implement a form of wear-leveling and garbage collection. This requires some work, and as a result, a DBMS can never obtain an IO throughput strictly equal to the throughput of the underlying flash chips even when the FTL guarantees optimal mapping.

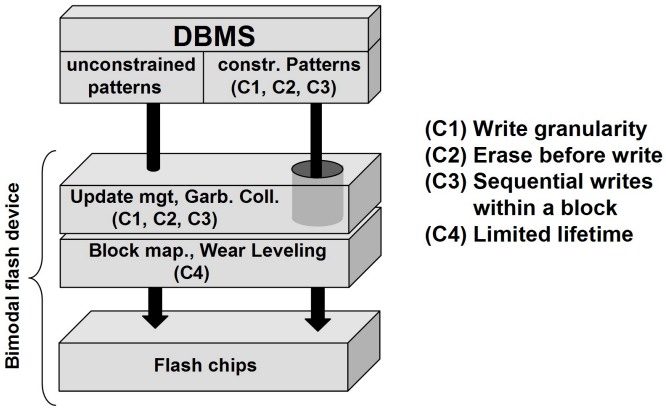


Figure 2: Bimodal FTL that combines near-optimal performance for IO as long as the DBMS respects constraints C1-C3, and best effort performance for other IO patterns where the FTL must enforce these constraints.

logical blocks for which the DBMS guarantees that writes are performed sequentially (C3) at the granularity of a flash page (C1), while any update is preceded by a Trim command (C2). The other logical blocks—on which all IO patterns are allowed—are mapped to one or more physical flash blocks, depending on the algorithms used by the FTL to manage constraints C1-C3. The design of the FTL optimizations for non-optimal blocks is not in the scope of this paper. State of the art techniques can be used [16, 18, 19, 15, 13] as long as they do not directly interfere with optimal blocks. We argue the feasibility of this approach in Sections 4 and 5.

Obviously sequential writes will result in an optimal mapping. Interestingly, semi-random write, introduced in [21], i.e., random write IOs done in such a way that they can be mapped sequentially on different logical blocks, will also result in an optimal mapping [9]. Sequential reads and random reads on optimal blocks benefit equally from an optimal mapping. An interesting side effect is that an optimal block never needs to be garbage collected as the sequence of writes in an optimal mapping does not create obsolete space to be reclaimed. For non-optimal blocks, the goal of the garbage collector must be to tend towards optimal mapping.

3.3 Impact on DBMS design

Bimodal flash devices provide a stable and optimal basis for DBMS design. Even if more work is needed to investigate the actual impact of our approach, we can already make some preliminary observations

Existing work focused on making database writes sequential (e.g., Append and Pack [29], ReSSD [20], NIPU [32]) is today restricted to repairing the bad random write performance of low-end SSDs. Such a technique would be required to leverage the optimal performance of sequential writes in case of a bimodal FTL. Also, techniques designed for flash chips (e.g., Lazy adaptive trees [2], or Page-differential Logging [17]), which are not today adequate in the context of SSDs, could naturally be applied to bimodal FTLs since they would generate only optimal blocks. Other techniques based on hashing or sorting (e.g., online maintenance of very large random samples [21]) can today only be applied to those SSDS that support (a limited form of) semi-random writes. Such techniques as well as hash-join or sort-merge would clearly benefit from the performance of semi-random writes on a bimodal FTL as long as bukets are aligned on block boundaries. Finally, the FlashScan and FlashJoin algorithms proposed in [31], that aggressively make use of random read IOs of small granularities are today rather well supported in current SSDs, even if random reads are slower than sequential reads on many SSDs. A bimodal FTL could ensure optimal performance with random reads as performant as sequential reads.

While it is impossible to develop a performance model of existing SSDs (because of their opacity and complexity), it is possible to envisage both analytic models and simulation models of bimodal flash devices, in order to explore the DBMS design space.

4. NARROW BIMODAL FLASH DEVICES

Now, let us focus on how we can design bimodal flash devices. Basically, the question is the following: Is it possible to implement a bimodal FTL without violating the constraints of a block device interface, i.e., fixed size IOs, flat name space of logical addresses, interface reduced to read/write/trim commands? This question boils down to (a) How to represent logical blocks and pages with a block device interface, (b) How can the FTL detect that the DBMS is submitting IOs that respect constraints C1-C3, and (c) How to guarantee optimal mapping in this case?

4.1 Bimodal Design

We propose an obvious implicit and immutable scheme for associating logical addresses to logical blocks and pages. The flash device must expose two constants[10]: *Logical Block Size* or LBS and *Logical Page Size* or LPS. LBS (resp. LPS) correspond to the size, in bytes of a flash block (resp. a flash page)[11]. For a logical address A, the logical block number LBN and logical page number LPN are obtained using the following trivial formulas: $LBN = A/LBS$ and $LPN = (A - LBN \times LBS)/LPS$, where / is the integer division.

While page size IOs are submitted sequentially within a logical block (starting from page 0), flash pages are written in the order the IOs are submitted. This way, the layout of flash pages within a flash block is optimal, and each read (either sequential or random) can be trivially mapped to an offset within a flash block. This is detected by maintaining in the safe cache, for each flash block, the physical position of the last write within the block. A bit indicates whether the mapping is optimal or not (initially, free blocks are optimal). We thus maintain in the controller cache a mapping with 4 bytes per block (22 bits for physical block id + 6-8 bits for current position within block (64-256 pages per blocks) + 1 bit flag for the optimal mapping).

An optimal block might become non-optimal if the submitted IOs are no longer sequential (e.g., updates, unaligned

[9]For instance, filling in parallel several flash buckets (for e.g. hashing a relation) will result in semi-random writes. In our approach, it is the number of logical blocks that limits the degree of parallelism in the semi-random writes.

[10]Such constants can be retrieved by the DBMS using specific command like GetDriveGeometry

[11]Parallelism within the flash device might lead to define LBS (resp. LPS) as a multiple of the flash block size (resp. the flash page size.

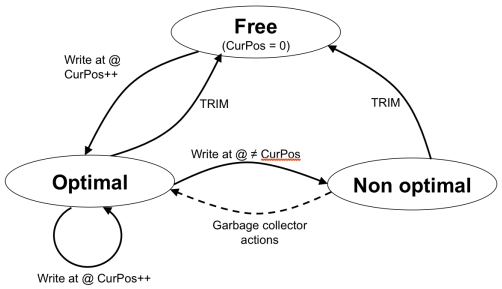


Figure 3: The states of a logical block and corresponding actions (transitions).

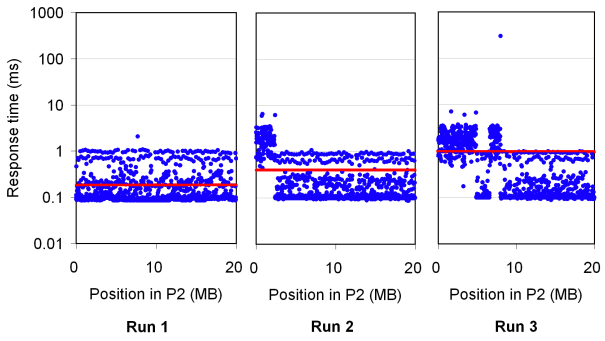


Figure 4: Response time (log scale) for each individual IOs (16KB) on partition P2 for runs 1, 2 and 3. The solid horizontal line represents the mean value.

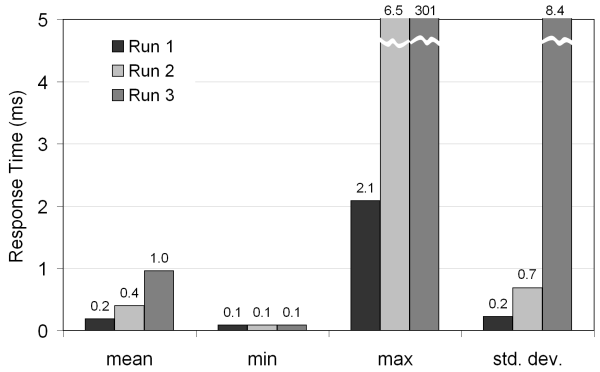


Figure 5: The graph shows the mean, min, max and standard deviation of response time for the sequential writes on P2 for each run. The Intel X25-E is not bimodal because the cost of sequential writes is not constant across runs.

IOs across page and block boundaries, random IOs). Conversely, classical garbage collector algorithms [19, 15], triggered in case of updates or random writes, will tend to convert non-optimal blocks into optimal ones. Again, many existing techniques can be used to mitigate the effects of random writes or updates on non-optimal blocks [16, 18, 19, 15, 13]. Figure 3 schematize the behavior of a single logical block in a bimodal FTL.

The obvious question now is whether any of the currently available flash devices implement such a FTL.

4.2 Are Existing Flash Devices Bimodal?

We can almost answer this question by looking at the data sheets of existing devices. First, no device explicitly provides a notion of logical block – blocks are abstracted away at the narrow interface. In practice, devices might silently implement the design we propose above. So, this is inconclusive. Second, only few devices actually provide a safe cache (e.g., Memoright does, Intel X25 does not). A DBMS relying on a flash device without safe cache must choose between performance (using the cache) and durability (write through the cache). Indeed, in our experience, disabling the cache typically results in an order of magnitude degradation of write performance (as neither mapping nor data is cached). Strictly speaking, a device might be bimodal even if it does not provide a safe cache. This is again inconclusive. Third, only the X25 devices from Intel support the TRIM command. This is more significant. Indeed, a flash device that does not support TRIM, does not allow the upper layers to respect constraint (C2), erase before write. Without TRIM, even a circular log structure as a database log will induce updates. As a result, only the X25 could be bimodal.

We devised a simple experiment to test whether the X25 is bimodal. The experiment consists in partitioning the logical address space in three (P1, P2, P3). Starting from a trimmed device, the experiment consists of two steps: (a) perform sequential writes (i.e., C1,C2 and C3 are enforced) on P2 and random writes on P1 and P3, and (b) Trim P2 (we make sure that trim is actually performed). Those two steps are repeated three times and we measure the response time for all sequential writes on P2. Note that we have observed long pauses after random writes on P1 and p3, to ensure that the device does not perform any background reorganization during sequential writes on P2.

If the device is bimodal, then the response time of sequential writes should remain constant for the three runs of sequential writes. Indeed, if the device is bimodal, then the large partition P2 should be mostly composed of optimal blocks for which all constraints have been enforced (there might be non optimal blocks at the boundaries of the partition due to the fact that logical block size and alignment is not documented).

Figure 4 shows detailed response time of each individual 16KB IO (log scale), while Figure 5 shows aggregated values (mean, min, max and standard deviation)[12]. We observe that response time and stability degrade with each run. The mean response time increases by a factor greater than 5 between Run 1 and Run 3. While the minimum response time remains approximately constant at 0.1 msec, the maximum increases from 2.1 msec in the first run to 301 msec in the third run. This experiment clearly demonstrates that the X25 is not bimodal. We can even say that a X25 equipped with a bimodal FTL should be able to achieve at most 0.1 msec mean response time for sequential writes on P2. Note that this experiment is a negative test. Defining a batteries of experiment to validate that a device indeed is bimodal is future work.

[12]The experiment was run on a 4-core Intel i5 processor running the flashIO utility we defined for the uFlip benchmark (see http://code.google.com/p/flashio/). The flash device used for the experiment is an X25-E 80GB.

5. RICH BIMODAL FLASH DEVICES

While a significant improvement with respect to today's devices, the narrow approach is not perfect: (i) Wear-leveling and garbage collection are performed without any knowledge of the stored data, (ii) The utilization of the controller cache is not optimized; (iii) A lot of complexity related to flash chips is managed at the DBMS level. Rich bimodal flash devices may address these shortcomings.

Extension of the block device interface have already been proposed in the context of flash devices [28, 23, 22] to manage complexity, control trade-offs and optimize embedded resources. While, Shu et al. [28] introduce the Data Management Set in the ATA protocol— connecting host and peripherals— that allows an application to communicate information about its I/O access behaviors to the underlying flash device, Rasjimwale et al [23] propose to use expressive interface such as object-based storage, and Prabhakaran et al. [22] focus on providing atomic writes. In the following, we describe how rich interfaces could be used to optimize the minimal FTL mode–most optimizations have already been presented in the literature–, and we quickly discuss non-optimal blocks. We deliberately avoid discussions of implementation or syntax issues, which are left for future works.

5.1 Optimal Blocks

From optimal blocks to optimal chunk: Since the block device interface does not provide any means to explicitly declare a set of optimal blocks (called hereafter *chunk*), these have to be detected by the FTL. More importantly, this detection leads to the management of a large number of small blocks, having an impact on the volume of metadata (mapping, statistics, and detection data) that must be stored into the safe cache. Explicit declaration of larger chunks may thus bring significant savings in terms of safe cache. For instance, a single 100 MB optimal chunk, explicitly declared to the flash device, e.g., for the log file of a DBMS, may bring a two order of magnitude reduction of the metadata wrt the implicit detection of 400 optimal blocks. While this optimization has no direct impact on the performance of optimal blocks (the mapping is already optimal), it allows for improved caching (see below).

Providing metadata: In a block device, wear leveling and garbage collection proceed blindly, without knowledge of the access patterns on the managed data. Choosing highly erased blocks to store data with low erase frequency and conversely will avoid useless blocks movements to balance erase counts. A similar strategy, based on detection, has been proposed in [10] in the context of a block device interface. Note that minimizing blocks movements increase performance and lifetime of the device.

Caching data: DBMS and FTL could collaborate to avoid producing non-optimal blocks for append data structures (e.g., log records, tables in append mode). The FTL can avoid update operations (and thus degrading an optimal block to non-optimal) if for each append structure, the last uncomplete written flash page can be kept inside the safe cache and only written to flash when it is completed.

Transmitting and caching payload: A rich flash device no longer has to be restricted to a flat, regular address space based on logical block addresses. We can consider write IOs which only transmit useful data, called hereafter the *payload*, i.e., fragments of pages as opposed to pages (note that we do not consider here that the flash device manages a representation of the database page layout). The main advantage of using payload instead of pages is that write operations can be buffered efficiently. (e.g., n inserted tuples in a database table will lead to buffer the aggregate size of the tuples plus some offsets-to recompose the pages-, compared with n IO sectors with a narrow interface!). Indeed, the safe cache is not polluted with data that already exists in the flash memory. Note that even if the safe cache is full, it may be more interesting to temporarily flush the payload part of the cache in a dedicated flash area (as a swap file) rather than flushing on their destination, in order to keep the destination blocks optimal. Write payload shares some ideas with the Page-Differential Logging approach [17].

Reading payload: Payload optimization is also very interesting for read operation since it avoids transmitting useless data from the chip to the controller and then to the host (typically, reading a 4KB page costs around 150 μs from which 125 μs are transmission costs through the serial interface from the chip to the controller). FlashScan [31] already mentioned in section 3.3 could benefit further from read payload, reading only subpages (i.e., corresponding to the ECC granularity).

Nameless writes: Nameless writes were introduced in [6]: the DBMS does not provide any destination address but let the FTL decides on the placement of the data and returns a handle for future reference. Nameless writes thus generates optimal blocks but are somehow easier to manage at the DBMS level, typically for temporary data.

Reorganization for free: A further improvement would consist in letting the FTL expose some aspects of the wear-leveling process to the DBMS to support additional optimizations. The wear leveling process sometimes has to move static data to balance the erase count across the whole device. In this case, the FTL reads a whole block that it rewrites on another physical location, thus bringing an opportunity to reorganize the block for free. The FTL could involve the DBMS in this process using a call-back mechanism in the way that external pagers interact with the operating systems [1]. Indeed, to allow using optimal blocks, DBMSs will probably resort to log-based approaches [2, 17] which might greatly benefit from such cleaning (almost) for free.

5.2 Non-optimal Blocks

While this paper focus on optimal blocks, there is in fact a greater potential for minimizing the important overheads generated by the management of non-optimal block: mapping cost (metadata management), garbage collection costs (with statistics like R/W frequency, expected pattern, etc.) and wear leveling costs (update frequency). For instance:

Hot-random-chunks and cold-random-chunks: The FTL can use different techniques for mapping different chunks whenever the DBMS associates access patterns to given chunks. The FTL can manage hot random chunks (i.e., frequent writes, scattered on the whole chunk) with a page mapping cached in the safe cache, while mapping can be performed at a larger granularity and on flash for cold random chunks. These strategies share the basic principles proposed in [13, 14], but are based on information delivered by the DBMS (and not detected by the FTL). Investigating further how a rich interface can benefit non-optimal block management is a topic for future work.

6. CONCLUSION

We argued that flash devices should provide guarantees to a DBMS so that it can devise stable and efficient IO management mechanisms. Based on the characteristics of flash chips, we defined a bimodal FTL that distinguishes between a mode where sequential writes, sequential reads and random reads are optimal while updates and random writes are forbidden, and a mode where updates and random writes are supported at the cost of sub-optimal IO performance. Interestingly, the guarantees of a minimal mode have been taken for granted in many articles from the database research literature. Our point with this paper is that these guarantees are not a law of nature, we must guide the evolution of flash devices so that they are enforced.

We described the design space for bimodal flash devices in the context of a block device interface, and in the context of a richer interface. Which one is most appropriate for a DBMS is an open issue. An important point is that providing optimal mapping guarantees does not hinder competition between flash device manufacturers. On the contrary, they can compete to (a) bring down the cost of optimal IO patterns (e.g., using parallelism), and (b) bring down the cost of non-optimal patterns without jeopardizing DBMS design. Future work includes designing and building a bimodal FTL in collaboration with a flash device manufacturer.

We can also derive a future work roadmap for database designers: (1) Define a performance model for bimodal flash devices in order to explore the DBMS design space. The reference here is the work of John Wilkes et al. on hard disks models [27]; (2) Explore the DBMS design space on top of narrow bimodal flash devices. The first challenge here is to compare and integrate the many ideas already proposed in the literature to establish a baseline. The interesting problem is then to optimize this baseline design; and (3) Explore the design space for the collaboration of DBMS and rich bimodal flash devices.

Finally, future work includes studying how operating systems as well as many data-intensive applications would benefit from flash devices with optimal mapping guarantees (e.g., warehouse scale distributed systems, game engines, IO-conscious algorithms).

7. ACKNOWLEDGMENTS

The authors wish to acknowledge Matias Bjørling for his precious help on experiments, Philippe Pucheral for his comments on earlier versions of this paper and Mehul Shah for inspiring discussions at SIGMOD 2010.

8. REFERENCES

[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *USENIX Summer*, 1986.

[2] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1), 2009.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX ATC*, 2008.

[4] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. L. Folgoc, B. Nguyen, P. Pucheral, I. Ray, I. Ray, and S. Yin. Secure Personal Data Servers: a Vision Paper. *PVLDB*, 3(1), 2010.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a Fast Array of Wimpy Nodes. In *ACM SOSP*, 2009.

[6] A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and V. Prabhakaran. Removing The Costs Of Indirection in Flash-based SSDs with Nameless Writes. In *USENIX HotStorage*, 2010.

[7] M. Bjørling, L. L. Folgoc, A. Mseddi, P. Bonnet, L. Bouganim, and B. Jónsson. Performing sound flash device measurements: some lessons from uFLIP. In *SIGMOD Conference*, 2010.

[8] M. Bjørling, P. Bonnet, L. Bouganim, and B. Jónsson. Understanding the Energy Consumption of Flash Devices with uFLIP. *IEEE Data Eng. Bull.*, to appear, 2010.

[9] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[10] L.-P. Chang and C.-D. Du. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1), 2009.

[11] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2), 2005.

[12] J. Gray. Tape is dead, disk is tape, flash is disk. `http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt`.

[13] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, 2009.

[14] J. Hu, H. Jiang, L. Tian, and L. Xu. PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD. *MASCOTS*, 2010.

[15] D. Jung, J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Trans. Embed. Comput. Syst.*, 9(4):1–41, 2010.

[16] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *Consumer Electronics, IEEE Transactions on*, 48(2), may. 2002.

[17] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song. Page-differential logging: an efficient and DBMS-independent approach for storing data into flash memory In *SIGMOD Conference*, 2010.

[18] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6), 2008.

[19] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), 2007.

[20] Y. Lee, J. Kim and S. Maeng. ReSSD: a software layer for resuscitating SSDs from poor small random write performance. In *SAC*, 2010.

[21] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1), 2010.

[22] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *OSDI*, 2008.

[23] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *USENIX ATC*, 2009.

[24] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6), 2001.

[25] S. W. Schlosser and G. R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? In *USENIX FAST*, 2004.

[26] S. W. Schlosser, J. Schindler, A. Ailamaki, and G. R. Ganger. Exposing and Exploiting Internal Parallelism in MEMS-based Storage. CMU Technical Report CMU-CS-03-125, 2003.

[27] E. A. M. Shriver, A. Merchant, and J. Wilkes. An Analytic Behavior Model for Disk Drives With Readahead Caches and Request Reordering. In *SIGMETRICS*, 1998.

[28] F. Shu and N. Obr. Data set management commands proposal for ATA8- ACS2. `http://www.t13.org/`, 2007.

[29] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMoN*, 2009.

[30] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7), 1981.

[31] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD Conference*, 2009.

[32] Y. Wang, K. Goda and M. Kitsuregawa. Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems. In *DEXA*, 2009.

Hyder – A Transactional Record Manager for Shared Flash

Philip A. Bernstein
Microsoft Corporation
philbe@microsoft.com

Colin W. Reid
Microsoft Corporation
colinre@microsoft.com

Sudipto Das†
University of California, Santa Barbara
sudipto@cs.ucsb.edu

ABSTRACT

Hyder supports reads and writes on indexed records within classical multi-step transactions. It is designed to run on a cluster of servers that have shared access to a large pool of network-addressable raw flash chips. The flash chips store the indexed records as a multiversion log-structured database. Log-structuring leverages the high random I/O rate of flash and automatically wear-levels it. Hyder uses a data-sharing architecture that scales out without partitioning the database or application. Each transaction executes on a snapshot, logs its updates in one record, and broadcasts the log record to all servers. Each server rolls forward the log against its locally-cached partial-copy of the last committed state, using optimistic concurrency control to determine whether each transaction commits. This paper explains the architecture of the overall system and its three main components: the log, the index, and the roll-forward algorithm. Simulations and prototype measurements are presented that show Hyder can scale out to support high transaction rates.

1. INTRODUCTION

The hardware platform for database systems is undergoing major changes with the advent of solid-state storage devices, high-speed data center networks, large main memories, and multi-core processors. These changes enable new database software architectures. This paper explores one such architecture: a log-structured multiversion database, stored in flash memory, and shared by many multi-core servers over a data center network. The architecture, shown in Figure 1, is embodied in a prototype system called Hyder.

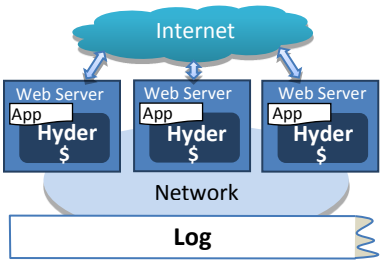


Figure 1 The Hyder architecture

Hyder is a transactional indexed-record manager. It supports read and write operations on indexed records within classical multi-step transactions. This is the basic functionality of the storage engine of a SQL database system. Hyder is designed to run on a cluster of servers that have shared access to a large pool of network-addressable storage, commonly known as a data-sharing architecture. Ideally, the storage is comprised of raw flash chips, although solid-state disks and possibly hard disks could work as well. Its main feature is that it scales out without partitioning the database or application. It is therefore well-suited to a data center environment, where scaling out is important and where specialized flash hardware and networking can be cost-effective.

† Work performed while employed at Microsoft Research.

5th Biennial Conference on Innovative Data Systems Research (*CIDR '11*) January 9-12, 2011, Asilomar, California, USA

1.1 Today's Alternative to Hyder

To understand the value of Hyder's no-partition scale-out feature, consider today's alternative: a data center architecture for database-based services, shown in Figure 2. The database is partitioned across multiple servers. The parts of the application that make frequent access to the database are encapsulated in stored procedures. The rest of the application runs in servers, either co-located with the web server or in a separate layer of servers (as shown). The application servers usually have a cache (denoted "$" in the figure) to minimize accesses to the database. Often, the application servers are partitioned, so their caches can be partitioned, thus enabling more of the database to be cached. Typically the application is responsible to choose which data to cache, to refresh this cache periodically, and to maintain cache coherence across the servers.

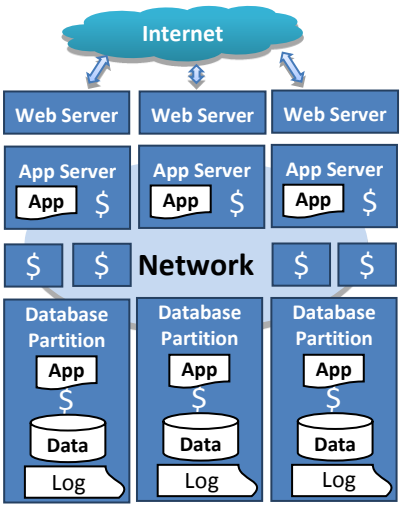


Figure 2 Scale-out architecture with partitioning

Some data that needs to be cached cannot be partitioned, such as a many-to-many relationship that is traversed in both directions. The friend-status relation for social networking is a well-known example. Such data is stored in separate cache servers.

Designing such systems is hard and requires special skills. The designer needs to choose a partition strategy that balances the load across servers, and minimizes or avoids distributed transactions (primarily due to the expense of two-phase commit). Migration mechanisms are needed to enable the system to grow by splitting and relocating overloaded partitions. And the application programmer needs to split application logic between the layers of servers, which implies distributed debugging.

1.2 Benefits of Hyder Architecture

Hyder simplifies application design by avoiding partitioning, distributed programming, layers of caching, and load balancing. Since it uses a data-sharing architecture, all servers can read from and write to the entire database (see Figure 1). This enables the database software to run in the application process, which simplifies application development by avoiding any distributed programming. It also avoids the expense of remote procedure calls

between the application and database. Since each server caches data it recently accessed or updated, the content of each server's cache reflects the accesses of transactions that ran recently on that server. So there is no need to partition the cache, either across servers of a given type or across layers of processes, because there are no layers. And since any transaction can execute on any server, load balancing is simply a matter of directing each transaction request to a lightly-loaded server.

In Hyder, each update transaction executes on one machine and writes to one shared log. Hence, it does not require two-phase commit. This saves message delays. It also avoids two-phase commit's blocking behavior, which requires operator intervention or heuristic decisions, both of which are undesirable, especially in a cloud-computing system.

Hyder simplifies cache coherence by using a multi-versioned database. This means there is no update-in-place. Therefore, although caches on different servers can have different versions of data, the caches are inherently coherent in that all copies of a given version are identical. Therefore, a query can be decomposed into subqueries that run against the same database version on different servers, e.g., to improve the response time of a query that accesses a lot of data. Moreover, each server continually and eagerly refreshes its cache, so it is rarely more than a few tenths of a second behind the last committed database state in the log.

Hyder scales out well without partitioning. Hyder's scaling limit of update transactions depends mostly on total update-transaction workload across all servers and on network and storage performance. Since there is no server-to-server communication, it does not depend on the number of servers.

Hyder's scale-out limits can be increased through careful application partitioning. However, most applications will never require it. For example, the effort to design a partitioned application is a wasted expense for a new application that never becomes popular, or whose popularity can be served by Hyder's scale-out limits. An application that experiences a few days of fame may need to scale out on short notice to avoid a success-disaster. Moreover, if an application does become popular quickly, Hyder can scale out without application redesign, thereby buying time for the application vendor to redesign for greater scale-out.

Some application scenarios that might benefit from Hyder's ability to scale out without partitioning are as follows:

- A cloud-based service for database applications may run tenants on a large cluster of servers. Most of these applications are likely to be small and can easily run on a single server, but some will need to scale out quickly.
- A packaged system that has a small number of servers can be purchased to run transaction processing applications. As usage increases, the system can grow incrementally by adding servers, without any software or database reconfiguration.
- An application that processes updates from social networks that form dynamically is not easily partitioned. Examples include multi-player games, real-time advertising of short-term sales events, and on-line news of a real-world disaster.

1.3 Hyder Software Layers

Data-sharing architectures are not new. They are supported by IBM DB2 Data Sharing [18], Oracle RAC [9], and Oracle (formerly DEC) Rdb [23]. Hyder differs from these systems in two ways. First, these systems usually require a soft partitioning of the applications, so that ownership of database pages does not have to move too frequently between servers. By contrast, Hyder requires no partitioning of applications to run well, though it can benefit from such partitioning. Second, Hyder uses a radically different architecture, with no lock manager. We therefore expect it to exhibit different performance tradeoffs than locking-based solutions; such a comparison is postponed as future work.

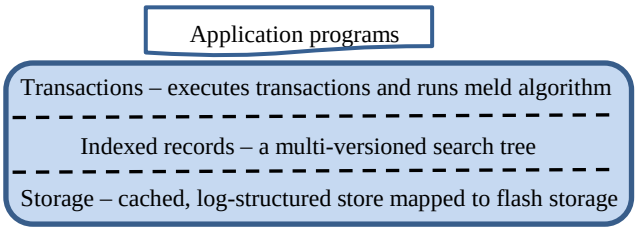


Figure 3 Hyder software layers

Hyder has the following three layers, shown in Figure 3:

- The storage layer offers a highly-available, load-balanced, self-managing, cached, log-structured store mapped to shared flash storage. The log *is* the database.
- The indexed record layer supports multi-versioned binary search trees mapped to log-structured storage.
- The transaction layer executes transactions. It runs a log roll-forward algorithm that continually refreshes the database cache. It uses optimistic concurrency control [20] to ensure transaction isolation.

This layering covers the functionality in the box labeled Hyder in Figure 1. There needs to be an application programming interface on top, such as SQL, but that is outside the scope of Hyder. It should be straightforward for Hyder to support an ISAM API or an API implementation that is layered on a narrow storage interface, such as that of MySQL [24]. Unfortunately, not all SQL database systems are so well modularized.

1.4 The Life of a Transaction

The life of an update transaction *T* is illustrated in Figure 4. *T* executes on one server, called *T*'s **executer**. When *T* starts, it is given the latest local copy of the database root, which defines a static snapshot of the entire database (step (1) in Figure 4). *T*'s updates are stored in a transaction-local cache. When *T* finishes executing, the after-images of its updates are gathered into a record called its **intention** (step (2)), which is broadcast to all servers (step (3)) and appended to the log (step (4)). For serializable isolation, *T*'s readset is included in the intention too.

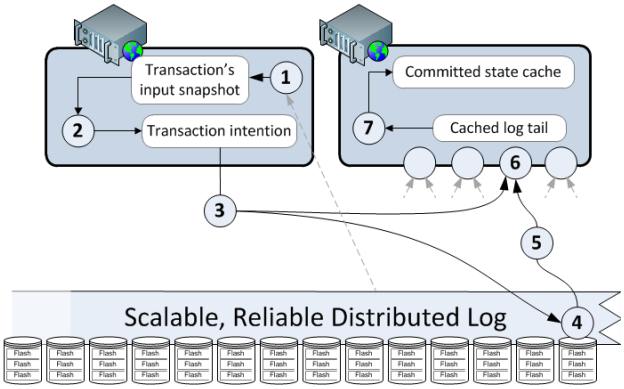


Figure 4 Steps in the life of a transaction

The log protocol ensures intentions are totally ordered, none are lost, and the offset of each intention is made known to all servers (step (5)). Since each server receives every intention and its offset, it can assemble a local copy of the tail of the log (step (6)). A server can detect if it failed to receive an intention from a hole in the sequence. In that case it can read the missing intention from the log.

Each server rolls forward the log on its cached partial-copy of the last committed state (step (7)). Unlike conventional database systems, appending *T*'s intention record *I* to the log does not commit *T*. Instead, when a server rolls forward *I*, it runs a procedure called **meld** that determines whether *T* actually committed. Conceptually speaking, *I* contains a reference *R* to *T's* **snapshot**, which is the last committed transaction in the log that contributed to the database state that *T* read (Figure 5). The intentions between *R* and *I* are called *T's* **conflict zone**. The meld procedure determines if any committed transaction in *T's* conflict zone includes an operation that conflicts with *T* with respect to *T's* isolation level. If there are no conflicts, then *T* is committed and the meld procedure merges *I* into the server's cached partial-copy of the database (the output of step (7)). Otherwise, *T* aborted. Since all servers (including *T's* executer) read the same log, they all make the same commit/abort decision regarding *T*.

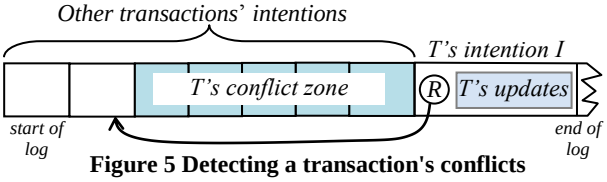


Figure 5 Detecting a transaction's conflicts

A transaction *T* executes in only one server (i.e., steps 1-3). However, all servers roll forward *T* using the meld procedure, including *T*'s executor. Thus, even *T*'s executor does not know whether *T* committed or aborted until after it melds *T*'s intention. At that point, it can notify *T's* outcome to *T's* caller.

Each server runs independently with no cross-talk to other servers. In particular, there is no lock manager and hence no locking bottleneck to constrain scale-out. Moreover, there is no two-phase commit. The only point of arbitration between servers is the atomic append of an intention to the log.

Of course, the architecture does have points of contention that can limit performance of update transactions: appending intentions to the log, broadcasting intentions to all servers, melding the log at each server, and aborting transactions due to conflicts. We quantify their effect in Section 5. Our measurements and simulations show that with today's technology, the system's throughput can reach 80K transactions per second (TPS) using a micro-benchmark of transactions with ten operations. Scale-out is practically unlimited for read-only transactions, since they do not consume any critical system resource; in particular, they are not melded. They can read a recent snapshot—the same snapshot at all servers, since all servers can access all versions of the whole database.

1.5 Hardware Trends

Hyder's architecture is enabled by four main hardware trends: high performance data center networks, large main memory, many-core processors, and solid state storage.

Networks – Commodity data center networks are 1 Gb/sec, with 10 Gb/sec available now and 40 and 100 Gb/sec already standardized. This enables many servers to share storage and broadcast the log with high performance. Network broadcasts are a bottleneck, but network bandwidth and latency will improve over time.

Main memory – At today's DRAM prices and with 64-bit processors, commodity servers can maintain huge in-memory caches. This reduces the rate at which servers need to access storage for cache misses.

Many-core – Given the declining cost of computation, Hyder can afford to squander computation by rolling forward the log on all servers to maintain consistent views across servers without server-to-server communication and by dedicating several cores per server to the meld activity.

Storage – Raw flash memory offers ~10^4 more I/O operations per second per gigabyte (GB) than hard disks. It costs ~100µs to read a 4KB page, ~200µs to write one, and there is no performance benefit to sequential access. This makes it feasible to spread the database across a log with less concern for physical contiguity than with hard disks. Solid-state disks (SSDs) are slower, but are improving. Sequential writes on some SSDs are already faster.

Although flash densities can double only two or three more times, other nonvolatile technologies are coming, notably phase-change memory (PCM). Instead of flash, Hyder might work with hard disks if servers have a large enough database cache to avoid too many cache misses, since they generate random reads, though some aspects of the storage layer would have to change.

Flash has two weaknesses that influence the Hyder design. First, pages cannot be destructively updated. A block of 64 pages must be erased before programming (writing) each page once. Erases are slow, ~2ms, and blocks other operations to a large portion of the chip. Second, flash has limited erase durability. MLC (cheap flash) can be erased ~10K times. For SLC (3x the price of MLC), it is ~100K times. Thus, flash needs wear-leveling, to ensure all pages are erased at about the same rate.

These two weaknesses are mitigated by Hyder's use of log-structured (i.e., append-only) storage, which allows the head of the log to be garbage-collected and cleaned while the tail is being written, and which spreads writes evenly across storage

By contrast, SSDs are not append-only; they support update-in-place. To support update-in-place while mitigating the weaknesses of flash, an SSD typically implements a log-structured file system [28]. This allows it to turn random page-writes into page-appends and automatically wear-levels the flash. This requires address mapping logic, garbage collection, and storage headroom. This functionality adds cost and can degrade performance, all in support of an update-in-place operation that Hyder does not need. Although Hyder can work well on SSDs, we believe it will perform better when implemented on raw flash. This is a classic end-to-end argument, where the application uses an append operation and the flash hardware works best as append-only, so there is no value in inserting an update-in-place operation in between.

1.6 Contributions

A short abstract about Hyder appeared in [4]. This paper expands that overview with descriptions of the following contributions:

- A fault-tolerant append-only log that offers an arbitration point between independent transaction servers.
- A log-structured multi-version binary-search-tree index
- An efficient meld algorithm that detects conflicts and merges committed updates into the last-committed state.

- A simulation analysis of the Hyder architecture under a variety of workloads and system configurations.

We describe Hyder's three layers bottom-up in Sections 2-4. Section 5 covers performance, Section 6 discusses related work, and Section 7 is the conclusion.

2. THE LOG

The log is comprised of multiple flash storage units, which we call **segments**. A segment could be a solid-state disk, a flash chip, or some other non-volatile solid-state storage component.

Each log record is a multi-page **stripe**, where each page of a stripe is written to a different segment. The pages of a stripe are written using a RAID-like erasure code, to enable recovery from a corrupted page, failed segment, or lost message.

A **failure zone** is a set of segments that can fail together due to a single-point failure, such as all of the segments in a rack (because they share a power supply and network connection). The best fault-tolerance is obtained when segments storing a stripe are all in different failure zones. A set of segments that store stripes is called a **stripe set**.

To enable a stripe set to be allocated in a dense sequence of segments, segments are addressed round-robin across failure zones. That is, they are assigned **segment IDs** such that if there are *n* failure zones, no two segments in any sequence of *n* segments are in the same failure zone. For example, if *n* = 8, then the first failure zone has segment IDs 0, 8, 16, ..., the second failure zone has segment IDs 1, 9, 17, …, etc. This arrangement ensures that the failure of a failure zone loses at most one page in any stripe.

2.1 Implementing AppendStripe

The log operations are AppendStripe and GetStripe. AppendStripe takes a stripe and stripe id as parameters and returns a **stripe reference**, which tells where the stripe was stored on each segment of the stripe set. AppendStripe is atomic. It is also idempotent, so a caller that fails to receive a reply from an AppendStripe can simply reissue the operation. GetStripe returns the stripe identified by a given stripe reference.

Log operations could be implemented by a server process or storage device that stores all pages of each stripe at the same offset of each segment. This requires the storage device to support an operation to write pages to a specific location, as is offered by SSDs. Given the widespread availability of SSDs, we believe such an implementation is worthwhile, as suggested in [2]. However, as discussed in Section 1.5, we believe that raw flash can offer better performance by mapping log-appends directly into storage-appends. We describe a log design for raw flash in this section.

To implement log operations on stripes, we propose to use a custom controller to access pages on flash storage using the operations AppendPage, GetPage, and EraseSegment. AppendPage appends a given page to a given segment and returns its address. GetPage returns a copy of the page stored at a given address. EraseSegment erases the entire content of the segment. A controller design that implements atomic and idempotent AppendPage and GetPage operations is sketched in [26].

The AppendPage operation is the only synchronization point between servers. If two servers append a page concurrently to the same segment, the segment's controller will serialize the operations and write the pages to successive storage locations. The usual technique of checksums on pages can be used to ensure that AppendPage is atomic.

If a stripe is comprised of *n* pages, then an AppendStripe operation is implemented by invoking *n* AppendPage operations on *n* segments in *n* different failure zones. Since failure zones are physically far apart, each segment must be updated via a different segment controller. Servers execute AppendStripe operations concurrently, which may execute in different orders in different controllers. Thus, in general, the pages of a stripe are not at the same offset of different segments of the stripe set. That is why AppendStripe returns a stripe reference and not a single offset.

If a log record is too large to fit in one stripe, then it needs to be split into multiple stripes. One of the log record's stripes contains the root of the log record. The other stripes can be appended to the log. Large, cold objects contained in the log record can be stored on hard disks, which are cheaper per-GB. All log-record data outside the root stripe must be written before the root stripe, so their addresses can be included in the root stripe. As we will see in Section 3, the log record is structured as a tree. Thus, the root stripe contains an upper portion of the tree that includes the root and points to subtrees that are stored in other stripes.

Each log record is a stripe, and log records must be totally ordered. This order is not self-evident from the order of a stripe's pages, since the order of its pages is different on different segments. To resolve this ambiguity, we define the relative order of stripes by the order of their pages in the first segment of the stripe set. This segment is called the **edge** of the stripe set.

Since a failure may prevent all pages of a stripe to be appended to its stripe set, some bookkeeping is required for servers to agree whether or not all of the pages of a stripe were written. This is done by maintaining a persistent log of stripe references, called the **end-write log**. After the AppendStripe operation receives acknowledgments for all of its corresponding AppendPage operations, it appends to the end-write log an **end-write record** that contains the stripe reference for the appended stripe. For fault tolerance, the end-write record must be appended to multiple segments. The choice of the number of segments should be based on an analysis of flash device failure rates and end-write log recovery time. We expect three will be enough. For multi-stripe log records, a server should wait to receive end-write acknowledgments for all of the non-root stripes before it appends the root stripe, to ensure that the root stripe has no dangling references.

Each log record is broadcast to all servers. First, the stripe is broadcast. After the stripe's end-write has been written, it too is broadcast so that servers know the stripe was successfully written.

The number of physical messages that are broadcast depends on several factors: the size of a log record, the size of a network packet, and the amount of batching of records into packets. In particular, end-writes are small, so many can be stored in a packet, at the cost of some latency to wait until the packet fills.

2.2 Failure Handling

There are three types of failures to consider: message loss, server failure, and segment failure. A short preview of our solution is in [2]. We discuss some basic cases here. For simplicity of exposition, we assume the log record fits in one stripe.

In many cases, lost messages can be retrieved by accessing the log. For example, if a server receives an end-write for a stripe but not all of the pages that the end-write references, then it can retrieve the missing pages from the log. If it receives a partial log

record that does not include an edge page, then it can simply ignore the record (unless and until the edge page shows up). If it detects a hole in the sequence of end-writes it has received, it should read the missing page(s) from an end-write segment.

However, what if it receives an edge page E, but does not receive an end-write for a stripe that includes E (within a timeout period)? Edge page E has reserved a slot in the log-record sequence, so later log records cannot be processed until it is known whether E's stripe will show up. Therefore, if it does not receive the rest of E's log record with a timeout period, it declares a log failure and initiates a recovery protocol.

Our recovery protocol is a variation on Vertical Paxos [22]. It seals the edge segment from further appends (e.g., set the segment's class to "sealed," as described below) and runs a consensus protocol to reach agreement on which stripes near the end of the log are complete and should be included. It then creates a new configuration of segments for the log and resumes normal operation. A similar process is used to cope with a permanent segment failure. There are many details to consider, which will be the subject of a future paper.

2.3 Sliding Window Striping

Allowing multiple servers to write stripes independently introduces some storage management problems. First, since flash storage is not cheap per-GB, variable-length stripes should be supported. This implies that initial segments of a stripe set will fill up faster than later ones. When the initial segment fills up, a new configuration must be allocated and all servers must agree to it.

Second, all servers have to agree on which segments are being used for each type of data: end-write records, totally-ordered root log stripes, and unordered non-root log stripes.

To coordinate server agreement, each controller maintains a persistent integer state-variable for each segment it manages, called its **class**. The integer value describes the type of data that can currently be appended to the segment. For concreteness, let us use zero for empty, one for a segment that stores pages of unordered (i.e., non-root) log stripes, two for ordered (i.e., root) log stripes, three for end-writes, and four for "sealed" (i.e., where no more appends are possible).

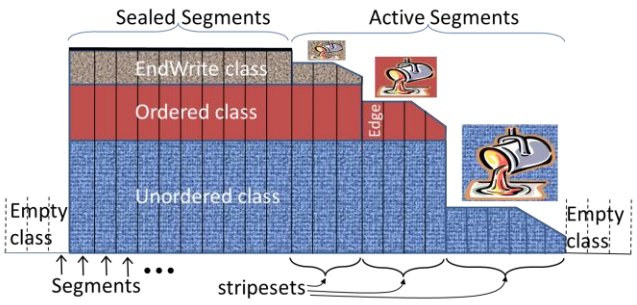


Figure 6 Sliding-window striping

We assign stripe sets to successive segment ranges in decreasing order of class. For example, there might be 3 segments for end-write stripes, then 4 for root stripes, and then 6 for non-root stripes (see Figure 6). Segments before the first end-write segment are sealed and those after the last non-root segment are empty.

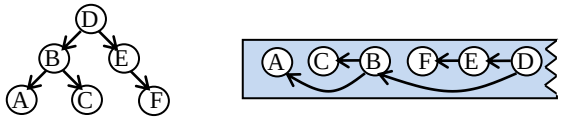
A class variable is added as a parameter to the AppendStripe operation. AppendStripe(C, S) for class C and segment S behaves as follows. If C < class(S), then do not perform the append and return an exception to the caller. If C = class(S) (i.e., the class of S), then perform the append. If C > class(S), then set class(S) to C and perform the append. This mechanism is used to ensure that all servers agree on which type of data to append to each segment.

Suppose the first segment of the end-write class fills up. Then the end-write stripe set would **advance**, thereby "capturing" the first segment (i.e., edge) of the ordered class (see Figure 6). The ordered class is now short by one segment, so it advances, capturing the first segment of the unordered class, and so on. In this sense, the stripe sets behave like sliding windows over the segments.

Higher layers of the system are responsible for garbage collection by copying reachable data in the first sealed segment to the end of storage and ensuring there is an ample supply of empty segments.

3. INDEX STRUCTURE

The index layer of Hyder stores the database as a search tree, where each node is a <key, payload> pair. For concreteness, we use a binary search tree in this paper. The tree is marshaled into the log (see Figure 7). Its basic operations are get, insert, delete and update of nodes and ranges of nodes, identified by their keys. The index layer also maintains a node cache of recently accessed parts of the tree.

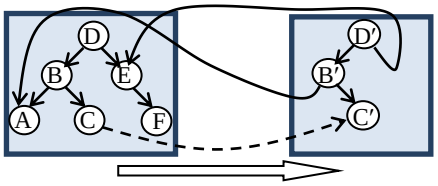


Binary search tree — Search tree marshaled into log

Figure 7 Marshaling a search tree into the log

If the tree stores a relational database, the key would be composite, beginning with the ID of a database, followed (for example) by sub-schemas, tables, and key values. Nodes in upper levels of the tree would span the set of databases. Their descendants would span sub-schemas, then tables, and then rows with key values. Each table can have secondary (i.e., non-clustered) indices, each of which is a table that maps a secondary key to the primary keys of rows that contain that secondary key, as is done in Microsoft SQL Server. Prefix and suffix truncation should be used to conserve space.

A node of the tree cannot be updated in place. To modify a node *n*, a new copy *n′* of *n* is created. Since *n*'s parent *p* needs to be updated with a new pointer to *n′*, a new copy *p′* of *p* is needed. And so on, up the tree. An example is shown in Figure 8. Notice that D′ is the root of a complete tree, which shares the unmodified parts of the tree with its previous version, rooted at D.



To update C's value, create a new version C′ and replace C's ancestors up to the root.

Figure 8 Copy-on-write

Insert and delete operations may trigger tree rebalancing, which updates more nodes than those on the path to the node being inserted or deleted. It may therefore be beneficial to rebalance periodically, rather than on every insert and delete.

An updated tree is logged in a transaction's intention. For good performance, it is important to minimize its size. For this reason, binary trees are a good choice. A binary tree over a billion keys has depth 30. A similar number of keys can be indexed by a four-layer B-tree with 200-key pages. But an update of that B-tree creates a new version of four pages comprising the root-to-leaf path, which consumes much more space than a 30-node path.

On the other hand, binary trees are known to have poor processor cache performance. In this respect, B-trees perform better and therefore may be preferable for a read-intensive workload, where it is important to optimize processor performance. For update-heavy workloads, we are experimenting with B+-trees that have low fanout and use prefix and suffix key-compression. Initial results suggest it might be able to generate intentions whose size is competitive with binary trees. B-trees also may be preferable in an implementation over hard disks, to reduce the random seeks, especially for sequential access in key-order. A comparative evaluation of these tradeoffs would be a worthwhile investigation. So would an evaluation of the many known optimizations for improving the processor performance of binary trees [19], since those optimizations may be less effective on trees that are fragmented in intentions spread across a log.

To simplify garbage collection of the log, each node pointer includes the segment ID of its earliest reachable segment. This is easy to maintain, as each new node simply picks the minimum of its two children. A segment that is older than any segment pointed to by a node is garbage. A simple tree traversal can identify all nodes in the oldest segment. They can be copied to the end of the log by a copier transaction, thereby freeing up the segment for reuse. Since the copier does not update the data it is copying, a write-conflict with an active transaction does not cause it to abort.

4. MELD

The transaction layer has two components. The executor runs new transactions, as described in the beginning of Section 1.4. The second component is the meld procedure, which is described here.

The meld procedure detects whether an intention experienced a conflict, and if not, merges its updates into the local copy of **the last committed state** (**LCS**). Rather than scanning the intention's conflict zone to determine whether the transaction experienced a conflict, meld is made more efficient by maintaining metadata in LCS that is sufficient to detect conflicts accurately. Thus, meld takes an intention and its server's LCS as input, and it returns a new version of LCS. That is, meld is a function—non-destructive with no side effects.

Meld is optimized further by not having to consider every item in the readset and writeset. It stops looking for conflicts as soon as it encounters a subtree of an intention whose corresponding LCS subtree did not change while the transaction executed. The speed-up from this optimization is significant for transactions that operate on non-hot data. In particular, it implies that for multi-stripe log records, meld often only needs to read the root stripe to process the log record's intention.

Meld must be deterministic. That is, it must produce exactly the same sequence of states on all servers. Otherwise, an intention generated by a transaction on one server might not be properly interpreted by meld running on another server. Hence, meld runs sequentially, processing intentions in log order. Meld can be parallelized to some extent, at least by parsing log records into objects on one thread before interpreting the log records (i.e., the roll forward activity) on another thread.

The update-transaction throughput across all servers is limited by the speed of meld. Therefore, meld must be fast. This is especially important because single-threaded processor performance is not expected to improve much for some time. Our current implementation can meld up to 400K TPS for small transactions.

A **serial intention** is an intention whose conflict zone is empty. That is, its snapshot is the transaction that immediately precedes it in the log. In that case, meld is trivial, since the intention's root defines its output's LCS. This case arises only when the update load is very light—when the transaction inter-arrival time is more than the end-to-end processing time of a transaction.

Melding a non-serial intention I is more complex, because LCS includes updates that were not in I's snapshot. Meld must check that these updates do not conflict with I, and if they do not, then it must merge I's updates into LCS. That is, it cannot simply replace LCS by I, as in the serial case.

For example, suppose transaction T_1 executes on an empty database, inserting nodes B, C, D, and E. (See Figure 9.) Then transactions T_2 and T_3 execute on T_1's output. T_2 inserts node A, and T_3 inserts F. T_2 and T_3 do not conflict, so after melding them LCS should include both of their updates.

Each node n in an intention I has a unique **version number** (**VN**), which is calculated based on the number of updated descendants in I and the LCS version against which I is melded. It also has a **source content VN** (**SCV**) that refers to the previous version of the node (i.e., the version in I's snapshot), and a flag **DependsOn** that is TRUE if I depends on n not having changed during $T(I)$'s execution. We denote node n in I_j by n_j. We denote its VN, SCV, and DependsOn flag by VN(n_j), SCV(n_j), and DependsOn(n_j). Similarly, VN of node n in the LCS is denoted VN(n_{LCS}).

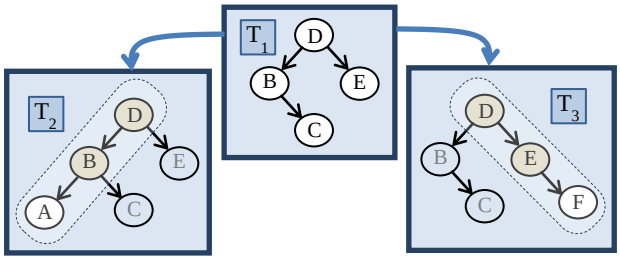


Figure 9 Example transactions to be melded

The need for DependsOn arises in part because some nodes in I were updated only because a descendant was updated. For example, in Figure 8, B′ was updated only because C′ was updated. In this case, DependsOn(C′) = TRUE while DependsOn(B′) = FALSE.

VN, SCV, and DependsOn enable meld to detect conflicts. If SCV(n_I) ≠ VN(n_{LCS}), then a committed transaction modified n while $T(I)$ was executing. In this case, if DependsOn(n_I) = TRUE, then $T(I)$ experienced a conflict and should be aborted.

This is an oversimplification, because it might be that SCV(n_I) ≠ VN(n_{LCS}) only because one of n's descendants was updated, not because n's content changed. If DependsOn(n_I) = TRUE because I depends on the content of n, then such an update does not constitute a conflict, since n's content in LCS has not changed. On the other hand, if DependsOn(n_I) = TRUE because $T(I)$ read the entire key range rooted at n and $T(I)$ uses serializable isolation, then SCV(n_I) ≠ VN(n_{LCS}) does imply a conflict since the content of

some record in the subtree rooted at *n* changed. By enriching the definition of SCV(*n*) and adding other metadata to *n*, we can distinguish these cases, but for pedagogical simplicity we will not make this distinction here. Details are in [5].

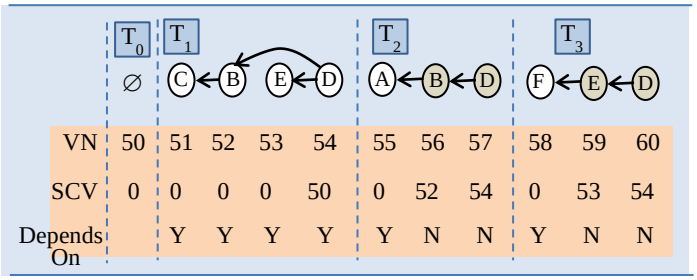


Figure 10 Log of the transactions of Figure 6

Suppose transactions T_1-T_3 (from Figure 9) are sequenced in the log as shown in Figure 10**.** Meld processes the log as follows:

1. Meld deduces that T_1 is serial, because $SCV(D_1) = 50 = VN(D_{LCS})$, which means T_0 immediately precedes T_1. So it melds T_1 by returning a new LCS whose root is D_1, where $VN(D_1) = 54$. Notice that this step required examining D_1 but none of its descendants.
2. Similarly, meld deduces that T_2 is serial and returns a new LCS whose root is D_2, where $VN(D_2) = 57$.
3. For T_3, meld sees that $SCV(D_3) \neq VN(D_{LCS})$ (i.e., $54 \neq 57$). Since $DependsOn(D_3)$ = FALSE, these unequal VN's do not indicate a conflict. However, a descendant of D_3 might depend on a node in LCS that was updated in T_3's conflict zone. So meld has to drill deeper into I_3 by visiting E_3.
4. Meld sees that $SCV(E_3) = VN(E_{LCS}) = 53$. Thus, the subtree rooted at E did not change while T_3 was executing. So there is no conflict and meld can declare T_3 as committed.

In (1), (2), and (4) above, meld is able to truncate the traversal of *I*. This happens whenever meld encounters a subtree of the intention that did not change while its transaction was executing. This is a very important optimization, which significantly reduces the time to meld the intention compared to a naïve algorithm that tests every node in the intention for a conflict. For example, consider the performance benefit if nodes A and F were leaves of very long paths and meld were truncated high in the tree, like it was here.

Now that meld knows that T_3 committed, it has to merge T_3's updates into LCS. Unlike the serial cases of T_1 and T_2, it cannot simply return D_3 as the root of the new LCS, because $SCV(D_3) \neq VN(D_{LCS})$. This means that before melding I_3, LCS includes updates that are not in I_3, namely, T_2's insertion of node A. Therefore, meld must create a new copy D_ϕ of D that points to B_2 on the left and E_3 on the right.

Since every node must reside in some intention, meld creates a new intention that contains the new D. This is called an **ephemeral intention**, because it exists only in main memory. It is uniquely associated with the transaction that caused it to be created, in this case T_3, and logically commits immediately after T_3. In this case, it has just one node D_ϕ, but in general it can have many nodes.

An ephemeral intention's nodes, called **ephemeral nodes**, must have unique VN's, since they are needed by meld to process the next intention that follows it. To do this, meld dynamically assigns VNs to nodes when it processes each intention (as it does for nodes physically in the intention). Like all of meld's activities, the generation of ephemeral intentions, ephemeral nodes, and their VN's is done deterministically, so that all servers produce identical states against which new transactions can execute.

There are many other details of meld that are needed for more precise detection of conflicts (i.e., with no false positives), coverage of all isolation levels including proper handling of phantoms, optimized per-node and per-intention metadata to minimize the size of intention records, flushing of ephemeral nodes, checkpointing, server initialization and fast recovery, and garbage collection. These topics are covered in [5].

5. PERFORMANCE

We cannot report yet on the performance of an end-to-end prototype, because our implementation of the transaction and indexed record layers is not yet integrated with a flash-based implementation of the log. However, through a combination of measurements of implemented components, known hardware speeds, mathematical analysis, and fairly extensive simulations, we can predict how a complete implementation would perform.

5.1 Bottleneck Analysis

As mentioned in Section 1.4, Hyder has four potential bottlenecks: appending intentions to the log, broadcasting intentions to all servers, melding the log at each server, and aborting transactions due to conflicts. We discuss each one in turn.

Logging – Although the log could be stored on solid-state disks, raw flash chips with a custom controller are preferred. Custom hardware is quite feasible, given today's rapid innovation in flash board products and use of custom hardware in data centers and database appliances. One benefit of custom hardware is speed. For example, putting nonvolatile write buffers in front of 20 flash chips that are served round-robin would reduce the 200 µs write latency of raw flash to 10 µs. Since log-appends are a bottleneck, this speedup over SSDs can be very important. If each log stripe contains one intention, then 10 µs write latency implies a limit of 100K TPS. Batching multiple intentions per log stripe can improve throughput further.

Networking – Switched networks process point-to-point messages in parallel. Therefore, with large server buffer caches, reads will not significantly reduce throughput for Hyder. By contrast, broadcasts block the network—a major bottleneck. The switching-time of current network switches is under 1 µs for 10 Gb Ethernet. Interconnect, protocol, and distance delays add to that. On the other hand, technology improvements continue to reduce latency. Switching time on 40 Gbps Ethernet is already under 400 ns.

Meld – Our current meld implementation can meld up to 400K TPS for transactions with two operations, dropping to 130K TPS for transactions with eight operations. Although single-threaded processor performance is not expected to improve much, meld can be parallelized further, which should yield a several-fold speedup. We report on extensive meld performance experiments in [5].

Optimistic Concurrency Control – Like any concurrency control algorithm, the abort rate of Hyder's optimistic concurrency control depends on the fraction of concurrently-executing transactions that conflict [20]. In turn, this depends on the probability that two randomly-selected transactions conflict, and the average number of transactions that execute concurrently at any given time. For a given arrival rate of new transactions, the faster they execute, the fewer that execute concurrently, and hence the smaller their conflict zones and the lower their abort rate.

The worst case consists of concurrent transactions that read and write the same data. In that case, a serializable execution must be serial, so the maximum throughput is the inverse of execution time. For example, with 200 µs transaction latency as observed in our prototype, the maximum throughput on a single write-hot data item is 5K TPS. Assuming Poisson arrivals and exponential service times, a naïve analysis predicts throughput of ~1.6K TPS per write-hot data item. Clearly, it is beneficial to detect such high-conflict transactions and run them on one server, which serializes and batches the transactions to obtain higher throughput.

5.2 Simulation Analysis

We present a detailed analysis of Hyder's performance at high loads using different workloads, access patterns, and isolation levels. The first part of our analysis focuses on system performance when there are no resource bottlenecks. Later, we evaluate and analyze Hyder's behavior in the presence of high data contention and resource contention.

We developed a simulation for Hyder using a discrete event simulator. The simulation model has three main modules: compute nodes, a network, and a log, representing the components in the system as shown in Figure 1. The compute nodes execute transactions, maintain a cache of recently accessed database objects using a least-recently-used replacement policy, and run the meld procedure. Each node has limited processing capacity; we use a four-core processor.

We model the database as a set of integer keys. The combined size of the key and payload is set to 100 bytes.

Each transaction is comprised of a fixed number of reads and writes controlled by the read/write ratio. Keys accessed by a transaction depend on the access distribution: We use **hotspot** (x% operations accessing y% data) and **uniform access** distributions.

A transaction consumes 2 µs of latency for each read that is serviced from cache and 10 µs of latency for each write. A cache miss results in reading the missing intention from the log. Each update transaction uses two broadcasts to all servers: one to broadcast the intention and one for the log to broadcast the intention's offset.

We model the network as a switch that simulates network delays. The network allows concurrent unicast messages to different destinations while a broadcast ties up the entire network. We use a switching delay of 400 ns for a 1500 byte frame resulting in a broadcast throughput of 40 Gbps, assuming 80% utilization. A much higher unicast throughput is supported. If we used a 10 Gbps network instead, it would be the bottleneck. If 80% utilized, it would reduce throughput by 1/3, compared to 40Gbp at 30%.

The log simulates appends to the flash by a 10 µs delay. Each intention is appended within a single append latency. That is, we assume an intention fits in one log stripe.

The meld process performs explicit conflict detection by checking a new transaction for conflicts against the set of committed transactions in its conflict zone. We use serializable and snapshot isolation. Meld processing is simulated by a latency of 10 µs as measured in our prototype implementation of meld.

Under the configurations used, the peak capacity of the system is 100K TPS. We therefore use a peak offered load of 80K TPS to ensure the resources are appropriately exercised without overloading them. We set the database size to 1 million elements, the transaction size to 10 operations (8 reads and 2 writes), and the cache size to 300K elements, with serializable as the default isolation level.

We compute the intention record sizes as a function of the depth of the database tree and the number of operations in the transactions that must be included in the intention; serializable isolation requires both the readsets and writesets, while snapshot isolation requires only the writeset. If an intention does not fit into a network frame, we use multi-frame messages and assume that the network guarantees their in-order delivery.

We use the following convention for series names in the graphs: **Hot-x-y** refers to hotspot access patterns with x% operations accessing y% data items and **Uniform** represents uniform access patterns; **SI** and **SR** represent snapshot and serializable isolation.

5.2.1 Abundant Resources

Effect of Skew

To analyze the effect of skew in access patterns, we use hotspot and uniform access distributions. For hotspot distributions, we vary x from 80% to 95% and we vary y from 5% to 20%. In all experiments, the throughput increases linearly with the offered load and is almost equal to the offered load. Figure 11 plots throughput as a function of the offered load and access distributions using serializable isolation. The linear increase in throughput with offered load is evident from the linear fit curve on the throughput values. This linear increase is observed for all access distributions and isolation levels.

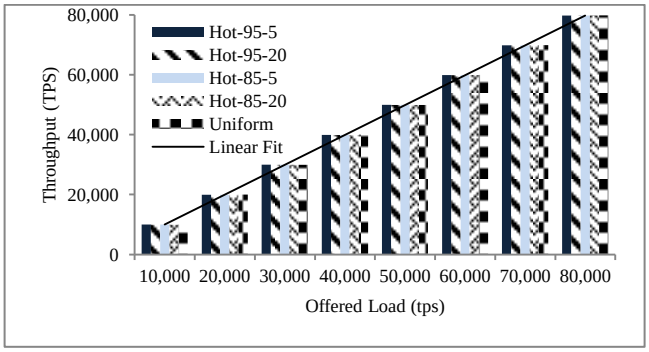


Figure 11 Transaction throughput (in TPS) as a function of the offered load and access distributions.

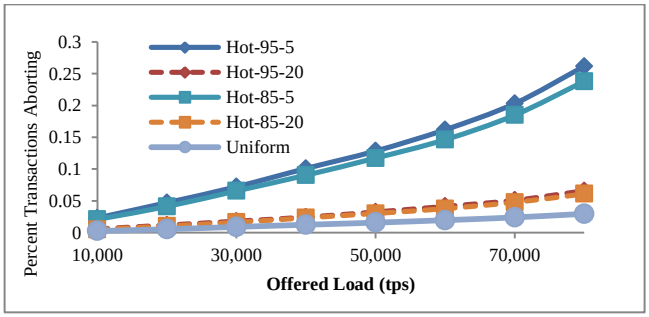


Figure 12 Percent transactions aborting as a function of offered load and access distributions.

Figure 12 plots the abort rate. It shows that although the abort rate increases with an increase in load, it is still negligibly small (in the range of 0.25%) at the peak offered load of 80K TPS. Though not shown, snapshot isolation has even lower abort rates, as expected. Furthermore, as long as the skew is not very high, the abort rate does not increase significantly with an increase in

offered load. This shows that given abundant network and flash capacity, the Hyder architecture can process a high volume of update transactions, and the throughput increases linearly with the offered load and is almost equal to the offered load.

We observed an average cache miss penalty of ~20 µs and an average transaction latency of ~50-60 µs for hotspot distributions and ~180 µs for uniform distributions. Due to the small cache miss rates for hotspot access distributions, the network throughput is dominated by broadcasts of intentions; peak network utilization for serializable isolation was ~10-12 Gbps, of which more than 90% was due to broadcasts. For uniform access distributions, unicast traffic is much higher as a result of cache misses; peak network utilization for serializable isolation was on the order of 65 Gbps, of which only 15% were broadcasts. Similarly, for hotspot access, less than 1% of the requests to the log are reads, while for uniform, this load rises to 80%.

The intention record size measured for serializable isolation was ~15.7 KB and for snapshot isolation was ~3.6 KB. The smaller intention size with snapshot isolation results in lower network load; for SI, network utilization peaks at about 3 Gbps and 15.5 Gbps for skewed and uniform distributions respectively with a share of broadcast and unicast traffic similar to that with serializable isolation.

Even at an offered load of 80K TPS, the mean conflict zone length was 7-10 intentions. This small conflict zone length is because transaction latencies were in the range of hundreds of microseconds. The conflict zone length also depends on the whether requests arrive in bursts or are spread out.

Effect of Cache Size

We now analyze the impact of cache hit rates on Hyder's performance. We vary the cache size from 50K to 300K items. At 50K items, the entire hot set does not fit in the cache (for the access patterns used in the experiments) resulting in a low cache hit rate, while at 300K elements, the hot set fits in the cache, resulting in a higher cache rate. A larger cache results in higher cache hit rates which reduce transaction latency, which in turn results in smaller conflict zones and lower abort rates.

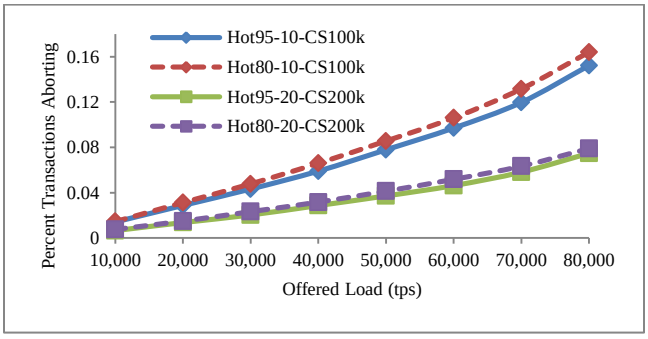


Figure 13 Percent transactions aborting as a function of offered load for different access distributions and cache sizes.

Figure 13 shows an interesting interplay between cache size and data skew and its impact on the abort rate. In each experiment, the cache size is set to the size of the hot set. For instance, a cache of 100K items is used for an access distribution where the hot set size is 10% of the database. The naming convention for the series is: "Access Distribution"-"Cache size"; thus, Hot95-10-CS100k refers to Hotspot distribution with 95% of operations accessing 10% of the data items, and with a cache size of 100K elements.

When the size of the hot set is kept constant and the percentage of operations accessing the cold set is increased, an increase in the abort rate is observed. Referring to Figure 13, the abort rate for the 80-10 access distribution is higher than that of the 95-10 distribution. This increased abort rate is counter-intuitive since a decrease in skew of the data items accessed should reduce the probability of a conflict. A closer analysis reveals that the 80-10 distribution has more accesses to the cold set. This increases the number of cache misses, which increases transaction latency, and in turn increases the conflict zone length and hence the conflict probability. This increase in transaction latency and conflict zone length is evident from Figure 14 which plots transaction latency as the primary vertical axis (on the left) and conflict zone length as the secondary vertical axis. This behavior is observed until the skew reduces to a point where the effect of higher conflict rate swamps the effect of shorter conflict zone. However, this counterintuitive behavior is not observed when the cache size is big enough to easily fit the hot set, as in Figure 12; the reason is that the larger cache is able to accommodate a small portion of the cold set and the entire hot set, which reduces the impact of skew on the cache hit rate and hence on the conflict zone length.

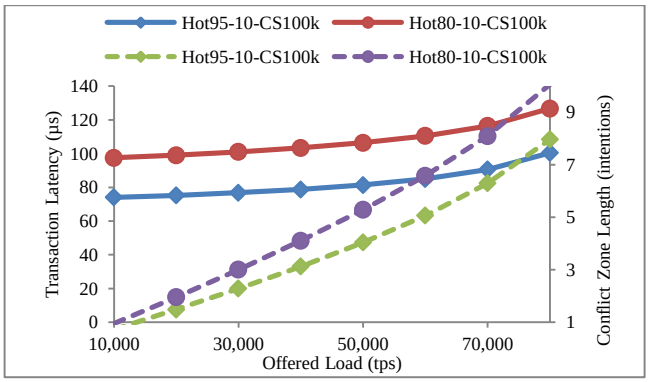


Figure 14 Transaction latency as a function of offered load for different access distributions and cache sizes.

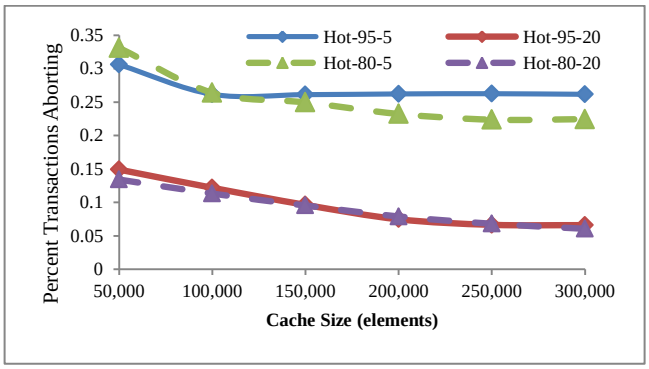


Figure 15 Percent transactions aborting as a function of cache size for different access distributions.

Figure 15 plots the abort rate as a function of cache size for different access distributions using serializable isolation; the offered load is kept constant at 80K TPS. As expected, as cache size increases the abort rate decreases, since more requests are served from the cache which results in shorter transaction latencies and conflict zones. When the cache size is 50K elements, it equals the hot set size for the 95-5 and 80-5 access distributions. We therefore observe behavior similar to Figure 13 where lower skew results in a higher abort rate.

For the access distributions 95-5 and 80-5, as the cache size grows, the entire hot set fits into the cache. Now the reduction in transaction latency resulting from higher skew is insignificant and the impact of higher skew on conflict probability dominates. As a result, when the cache is larger than the hot set, a higher abort rate is observed for a more skewed distribution; in Figure 15, the 95-5 distribution has a higher abort rate than the 80-5 distribution.

When the cache is too small to fit the hot set, a higher skew in access distribution does not result in a higher cache hit rate, since elements in the hot set also incur cache misses. As a result, the transaction latency, and hence the conflict zone length, for more skewed distributions are approximately equal to those of less skewed distributions. Therefore, the effect of skew dominates, resulting in a higher abort rate for skewed distributions. This behavior is evident from the experiments with 95-20 and 80-20 distributions where the hot set has 200K data items.

When the cache size is under 200K elements, the more skewed distribution (95-20) has a higher abort rate. When the cache size equals 200K elements, as expected, the 80-20 distribution experiences a marginally higher abort rate. Beyond 200K elements, the abort rate of 80-20 is again lower compared to that of 95-20. The cache size also impacts network utilization since cache misses result in unicast network messages to read the specified intention record from the log.

Effect of other parameters

For a given access distribution, increasing the number of reads reduces the abort rate. Similarly, increasing the transaction size increases the abort rate. Abort rates as high as 3.5% are observed for a skewed access distribution for transactions with 25 operations where the hot set comprises only 5% of the database. However, the abort rate decreases considerably for distributions with less skew, accompanied by a smaller gradient of increase.

We also experimented with a workload of variable-size transactions. As expected, increasing the average transaction size increases the abort rate. However, it is interesting that with variable-sized transactions whose average size is S, the abort rate is lower than with fixed-size transactions of size S. A closer analysis reveals that variable-size transactions broadcast fewer messages on average than fixed-size transactions, which reduces transaction latency, conflict zones, and hence conflict probability.

5.2.2 Data Contention

In the previous experiments, data contention was low and had an insignificant effect on throughput, even at the peak offered load of 80K TPS and with 95% of the operations accessing 5% of the database. Now, we focus on the performance of workloads with high data contention. We use smaller databases (10K to 100K), higher skew (99% of operations accessing 1% of the data items), more writes per transaction (1:1 read/write ratio), and larger transaction size. Figure 16 plots the throughput as a function of offered load. We use serializable isolation for all the experiments.

The first three series correspond to database sizes of 10K, 50K, and 100K and use transactions with 8 reads and 2 writes. The database size is kept constant of 100K for the remaining two experiments. The fourth series corresponds to transactions with 5 reads and 5 writes (1:1 read/write ratio), while the fifth series uses transactions with 16 reads and 4 writes.

We observe a significant impact of abort rate on throughput; the throughput curve plateaus out only for a database size of 10k or transaction size of 20 where abort rates are as high as 50-60%. As expected, the impact is much less for snapshot isolation, where no significant effect on throughput was observed (not shown).

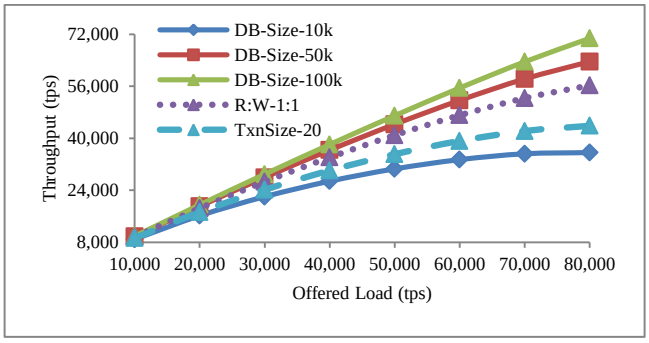


Figure 16 Transaction throughput (in tps) as a function of offered load for Serializable Isolation. Triangular markers correspond to Database size of 100k elements with different read/write ratios and transaction sizes.

5.2.3 Resource Contention

Pure data contention resulting from conflicting accesses does not significantly affect performance. However, heavy resource contention causes thrashing. For example, if the transaction execution rate reaches the maximum rate of log appends, throughput drops sharply (see Figure 17). Overloading the network or meld algorithm results in similar behavior. During an overload, transactions execute longer (latency of ~100 ms vs. ~200 µs under normal load), which increases the abort rate. Aborted transactions consume resources, which increases resource contention, resulting in a negative feedback loop. The effect is stronger for transactions with skewed access, because of their higher abort rate for a given conflict zone length.

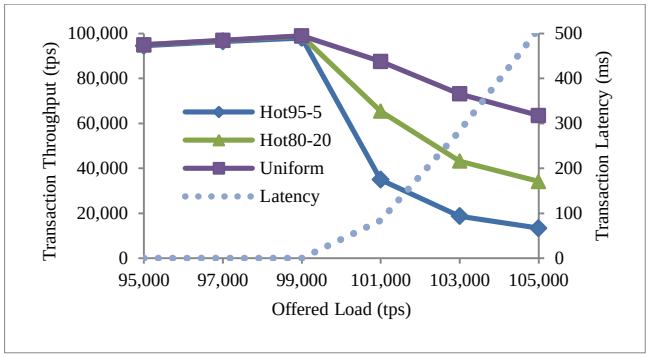


Figure 17 Thrashing resulting from resource contention.

Of course, Hyder should back off the workload when it detects thrashing. Moreover, it should do a trial meld before appending a transaction to the log. If it detects a conflict, it will not waste a broadcast slot and log append, thereby defeating the negative feedback loop, making the throughput drop-off much less steep.

5.3 Future Performance Work

Our simulation analysis shows that for practical workloads, such as highly-skewed hotspot access patterns and a database of 1M items, data contention induced by resource contention occurs long before "pure" data contention due to conflicts. Therefore, it is essential to use load-control to prevent resource contention. Moreover, to deal with very high conflict rates (as in Section 5.2.2), techniques for advanced transaction scheduling cognizant of conflict patterns will be useful. For example, a soft partitioning

of transactions that access write-hot data could enable them to be synchronized in the server before appending their intentions to the log. In turn, this will complicate load balancing, since the partitioning will constrain where such transactions should run.

The critical resources in Hyder—the network, log, and meld—have hard scaling limits imposed by the underlying hardware, which limits Hyder's scalability and peak capacity. Parallelism can be explored to increase the peak capacity of these critical resources. One such extension is to partition the database to enable parallel logging and melding. These extensions are worthwhile areas for future work.

6. RELATED WORK

Hyder is a data-sharing system. Other well-known data-sharing systems are Oracle RAC [9], Oracle Rdb [23] and IBM DB2 [18]. These systems use a shared lock manager to ensure that at most one server at a time has access to a page. If a server releases its write lock on a page, it must make the updated page available to other servers either by writing the page to disk or servicing reads from other servers. Thus, the granularity of data sharing between servers is a page, rather than an indexed record in Hyder. A failure that affects the lock table requires server coordination to recover, which is not needed in Hyder since servers are independent. On the other hand, the failure of a log segment in Hyder requires coordinated recovery, which is not required in systems that use server-local logs. These locking-based systems are known to require soft partitioning of the transaction load to avoid having pages ping-pong between servers. While Hyder can benefit from such partitioning, it performs well without it. The performance of these systems was studied in [25]. A comparison of their behavior to Hyder is suggested as future work.

There are many shared-nothing database systems designed for scale-out. The benefits of this architecture were demonstrated in the 1980s in many systems, such as Bubba, Gamma, Tandem, and Teradata. DeWitt and Gray [16] provide an excellent summary of that work. More recently, shared-nothing key-value stores have been developed for cloud computing with similar goals, such as Google's Bigtable [8], Amazon's Dynamo [15], Facebook's Cassandra [21], and Yahoo!'s PNUTS [10]. However, unlike Hyder, these systems do not support ACID transactions and they require the database to be partitioned. Microsoft's SQL Azure [3][6] and ElasTraS [14] are cloud-oriented database systems that do support ACID transactions but restrict each one to access one partition. The partitioned access requirement is relaxed somewhat in G-Store [13], since data access groups can be reformed dynamically.

A transactional, partitioned, key-value store that does not restrict a transaction's access pattern is described in [1]. Like Hyder, it uses optimistic concurrency control. Unlike Hyder, it is a shared-nothing system and hence requires two-phase commit. During its execution, a transaction tracks the versions of data it reads and writes. At commit time it sends this information to each server it accessed; the server either validates its readset and applies its writes as part of phase one, or detects a readset item changed and tells the transaction to abort. Among the key-value stores mentioned in the previous paragraph, its functionality is most similar to Hyder's, but its mechanisms are quite different. It would be interesting to compare their performance.

Another transactional, partitioned key-value store that uses optimistic concurrency control to synchronize transactions with two-phase commit for atomicity is ecStore [31].

Hyder strongly resembles a primary-copy replicated database, in the sense that it generates a single log and sends it to many servers, each of which applies updates to its local copy (i.e., a replica). However, Hyder is different from primary-copy replication in two ways. First, Hyder does not use a primary copy! Second, log records include the updates of both committed and aborted transactions. In most primary-copy replication systems, only committed transactions are sent to replicas. We do not know of any replication systems with the latter two properties. Gupta et al. [17] propose a replication mechanism similar to Hyder, but different in that it uses a single server as the serialization point and supports weaker forms of consistency that are sufficient for their application domain. A survey of recent work on primary-copy replication can be found in [7].

Multi-master replication, sometimes called optimistic replication, is similar to Hyder in that conflicting transactions can run to completion and their conflicts are detected later. Unlike Hyder, these mechanisms do not support standard isolation levels. Surveys of this work appear in [29] and [30].

The startup company RethinkDB has built a log-structured storage engine for MySQL, targeted for use on solid-state storage [27]. The database is append-only and "lock-free." However, they have not yet published details about their index structure or concurrency control algorithm. They support primary-copy replication, but are apparently not a data-sharing system.

Dan et al. [11] present an analytical model and simulation study of the effect of data and resource contention on transaction throughput for optimistic concurrency control (OCC) [20]. We observed similar behavior in our simulations: data contention alone causes system throughput to plateau out and resource contention causes thrashing. Transaction latency during high resource contention in Hyder also has behavior similar to that reported in [11].

The interplay of skew in data access distribution, cache hit ratio, and abort probability for lock-based concurrency control was studied by Dan et al. [12]. Our experimental study extends these results by demonstrating a similar interplay when using OCC. We further analyze the interplay of cache size with working set size for a hotspot distribution. Yu et al. [32] model the impact of improved cache hit rate for transactions restarted after an abort and study its effect on system throughput. If an aborted transaction restarts at the same compute node in Hyder, we expect to observe a similar behavior; however, we did not observe a significant impact since most of our simulations observed very low abort rates.

7. CONCLUSION

We have described the architecture and major algorithms of Hyder, a transactional record manager that scales out without partitioning. It uses a novel architecture: a log-structured multi-version database, stored in flash memory, and shared by many multi-core servers over a data center network. We demonstrated the feasibility of the architecture by describing two new mechanisms, a shared striped log and a meld algorithm for performing optimistic concurrency control and applying committed updates to each server's cached partial copy of the database. We presented performance measurements and simulations that demonstrate linear scalability up to the limits of the underlying hardware.

Many variations of the Hyder architecture and algorithms would be worth exploring. There may also be opportunities to use Hyder's logging and meld algorithms with some modification in other contexts, such as file systems and middleware. We

suggested a number of directions for future work throughout the paper. No doubt there are many other directions as well.

8. ACKNOWLEDGMENTS

This work has benefited from challenging discussions with many engineers at Microsoft over the past three years, too numerous to list here. We thank José Blakeley, Dave Campbell, Quentin Clark, Bill Gates, and Mike Zwilling for their early support of the project. We also thank our collaborators on the meld implementation, Ming Wu and Xinhao Yuan, and on the log implementation, Mahesh Balakrishnan, Dahlia Malkhi, and Vijayan Prabhakaran. We also thank Gustavo Alonso, Goetz Graefe, Dahlia Malkhi, Sergey Melnik, and Ming Wu for their reviews of early drafts of this paper and the anonymous referees for many excellent suggestions.

9. REFERENCES

[1] Aguilera, M.K., W.M. Golab, M.A. Shah: A practical scalable distributed B-tree. PVLDB 1(1): 598-609 (2008)

[2] Balakrishnan, M., P. Bernstein, D. Malkhi, V. Prabhakaran, and C. Reid: Brief Announcement: Flash-Log, A High Throughput Log. DISC 2010, LNCS 6343: 401-403

[3] Bernstein, P.A., I. Cseri, N, Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, T. Talius: Adapting Microsoft SQL Server for Cloud Computing. ICDE 2011, to appear.

[4] Bernstein, P.A. and C.W. Reid: Scaling Out Without Partitioning. HPTS 2009, 3 pp.

[5] Bernstein, P.A., C.W. Reid, M. Wu, X. Yuan: Optimistic Concurrency Control by Melding Trees. Submitted for publication.

[6] Campbell, D.G., G. Kakivaya, N. Ellis: Extreme scale with full SQL language support in Microsoft SQL Azure. SIGMOD 2010: 1021-1024.

[7] Cecchet, E., G. Candea, A. Ailamaki: Middleware-based database replication: the gaps between theory and practice. SIGMOD 2008: 739-752

[8] Chang, F., J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber: Bigtable: A Distributed Storage System for Structured Data. ACM TOCS 26(2): (2008).

[9] Chandrasekaran, S. and R. Bamford: Shared Cache - The Future of Parallel Databases. ICDE 2003: 840-850.

[10] Cooper, B.F., R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, R. Yerneni.: PNUTS: Yahoo!'s hosted data serving platform. PVLDB 1(2): 1277-1288 (2008).

[11] Dan, A., D.F. Towsley, W.H. Kohler: Modeling the Effects of Data and Resource Contention on the Performance of Optimistic Concurrency Control. ICDE 1988: 418-425.

[12] Dan, A., D.M. Dias, P.S. Yu: The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment. VLDB 1990: 419-431.

[13] Das, S., D. Agrawal, A. El Abbadi: G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. SOCC 2010: 163-174.

[14] Das, S., S. Agarwal, D. Agrawal, A. El Abbadi: ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. UCSB CS Tech Report 2010-04.

[15] DeCandia, G, D. Hastorun, M. Jampani, G. Kakulapati , A. Lakshman , A. Pilchin , S. Sivasubramanian , P. Vosshall , W. Vogels: Dynamo: Amazon's Highly Available Key-value Stor. Proc. 21st SOSP: 205-220 (2007).

[16] DeWitt, D.J and J. Gray: Parallel Database Systems: The Future of High Performance Database Systems. *CACM* 35(6): 85-98 (1992)

[17] Gupta, N., A.J. Demers, J. Gehrke, P. Unterbrunner, W.M. White: Scalability for Virtual Worlds. ICDE 2009:1311-1314

[18] Josten, J.W., C. Mohan, I. Narang, and J.Z. Teng. "DB2's Use of the Coupling Facility for Data Sharing." IBM Systems Journal 36(2): 327-351 (1997).

[19] Kim, C., J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, P. Dubey: FAST: fast architecture sensitive tree search on modern CPUs and GPUs. SIGMOD 2010: 339-350

[20] Kung, H. T. and J.T. Robinson: On Optimistic Methods for Concurrency Control. ACM TODS 6(2): 213-226 (1981).

[21] Lakshman, A. and P. Malik: Cassandra: a decentralized structured storage system. Operating Systems Review 44(2): 35-40 (2010)

[22] Lamport, L., D. Malkhi, L. Zhou: Vertical paxos and primary-backup replication. PODC 2009: 312-313.

[23] Lomet, D.B., R. Anderson, T.K. Rengarajan, and P. Spiro: How the Rdb/VMS Data Sharing System Became Fast. http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-92-4.html, Technical Report CRL 92/4 (May 1992).

[24] MySQL 5.5 Reference Manual, Chapter 13, http://dev.mysql.com/doc/refman/5.5/en/index.html

[25] Rahm, E., Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems. ACM TODS 18(2): 333-377 (1993)

[26] Reid, C.W. and P.A. Bernstein: Implementing an Append-Only Interface for Semiconductor Storage. IEEE Data Eng. Bulletin 33 (4) (Dec. 2010).

[27] RethinkDB, http://www.rethinkdb.com/.

[28] Rosenblum, M. and J.K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. 10(1): 26-52 (1992).

[29] Saito, Y. and M. Shapiro. Optimistic replication. ACM Computing Surveys 37(1): 42-81 (2005).

[30] Terry, D.B.: Replicated Data Management for Mobile Computing. Morgan & Claypool, 2008.

[31] Vo, H. T., C. Chen, B. C. Ooi: Towards Elastic Transactional Cloud Storage with Range Query Support. PVLDB 3(1) : 506-517 (Sept. 2010).

[32] Yu, P.S. and D.M. Dias: Impact of large memory on the performance of optimistic concurrency control schemes. Int'l Conf on Databases, Parallel Architectures and their applications. PARBASE-90: 86-90.

Rethinking Database Algorithms for Phase Change Memory

Shimin Chen
Intel Labs Pittsburgh
shimin.chen@intel.com

Phillip B. Gibbons
Intel Labs Pittsburgh
phillip.b.gibbons@intel.com

Suman Nath
Microsoft Research
sumann@microsoft.com

ABSTRACT

Phase change memory (PCM) is an emerging memory technology with many attractive features: it is non-volatile, byte-addressable, 2–4X denser than DRAM, and orders of magnitude better than NAND Flash in read latency, write latency, and write endurance. In the near future, PCM is expected to become a common component of the memory/storage hierarchy for a wide range of computer systems. In this paper, we describe the unique characteristics of PCM, and their potential impact on database system design. In particular, we present analytic metrics for PCM endurance, energy, and latency, and illustrate that current approaches for common database algorithms such as B$^+$-trees and Hash Joins are suboptimal for PCM. We present improved algorithms that reduce both execution time and energy on PCM while increasing write endurance.

1. INTRODUCTION

Phase change memory (PCM) [3, 10] is an emerging non-volatile memory technology with many attractive features. Compared to NAND Flash, PCM provides orders of magnitude better read latency, write latency and endurance,[1] and consumes significantly less read/write energy and idle power [9, 10]. It is byte-addressable, like DRAM memory, but consumes orders of magnitude less idle power than DRAM. PCM offers a significant density advantage over DRAM, which means more memory capacity for the same chip area and also implies that PCM is likely to be cheaper than DRAM when produced in mass market quantities [22]. While the first wave of PCM products target mobile handsets [24], in the near future PCM is expected to become a common component of the memory/storage hierarchy for laptops, PCs, and servers [9, 15, 22].

An important question, then, is *how should database systems be modified to best take advantage of this emerging trend towards PCM?* While there are several different proposals for how PCM will fit within the memory hierarchy [10] (as SATA/PCIe based data storage or DDR3/QPI based memory), recent computer architecture and systems studies all propose to incorporate PCM as the bulk of the system's main memory [9, 15, 22]. Thus, in the PCM-DB project [19], we are focusing on the use of PCM as the primary main memory for a database system. This paper highlights our initial findings and makes the following three contributions.

First, we describe the unique characteristics of PCM and its proposed use as the primary main memory (Section 2). Several attractive properties of PCM make it a natural candidate to replace or compliment battery-backed reliable memory for general database systems [18], and DRAM in main memory database systems [11]. However, a unique challenge arises in effectively using PCM: Compared to its read operations, PCM writes incur higher energy consumption, higher latency, lower bandwidth, and limited endurance. Therefore, we identify reducing PCM writes as an important design goal of PCM-friendly algorithms. Note that this is different from the goals of flash-friendly algorithms [2, 17], which include reducing the number of *erases* and *random writes* at much *coarser* granularity (e.g., 256KB erase blocks and 4KB flash pages).

Second, we present analytic metrics for PCM endurance, energy, and latency. While we believe that PCM may have a broad impact on database systems in general, this paper focuses on its impact on core database algorithms. In particular, we use these metrics to design PCM-friendly algorithms for two core database techniques, B$^+$-tree index and hash joins (Section 3). In a nutshell, these algorithms re-organize data structures and trade off an increase in PCM reads for reducing PCM writes, thereby achieving an overall improvement in all three metrics.

Third, we show experimentally, via a cycle-accurate X86-64 simulator enhanced with PCM support, that our new algorithms significantly outperform prior algorithms in terms of time, energy and endurance (Section 4), supporting our analytical results. Moreover, sensitivity analysis shows that the results hold for a wide range of PCM parameters.

The paper concludes by discussing related work (Section 5) and highlighting a few of the many interesting open research questions regarding the impact of PCM main memory on database systems (Section 6).

2. PHASE CHANGE MEMORY

In this section, we discuss PCM technology, its properties relative to other memory technologies, its proposed use as

[1] *(Write) endurance* is the maximum number of writes for each memory cell.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

Table 1: Comparison of memory technologies.

	DRAM	PCM	NAND Flash	HDD
Read energy	0.8 J/GB	1 J/GB	1.5 J/GB [28]	65 J/GB
Write energy	1.2 J/GB	6 J/GB	17.5 J/GB [28]	65 J/GB
Idle power	~100 mW/GB	~1 mW/GB	1–10 mW/GB	~10 W/TB
Endurance	∞	$10^6 - 10^8$	$10^4 - 10^5$	∞
Page size	64B	64B	4KB	512B
Page read latency	20-50ns	~ 50ns	~ 25 μs	~ 5 ms
Page write latency	20-50ns	~ 1 μs	~ 500 μs	~ 5 ms
Write bandwidth	~GB/s per die	50-100 MB/s per die	5-40 MB/s per die	~200MB/s per drive
Erase latency	N/A	N/A	~ 2 ms	N/A
Density	1×	$2-4\times$	4×	N/A

Note: The table contents are based mainly on [10, 15, 22].

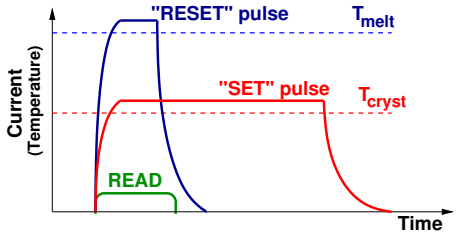


Figure 1: Currents and timings (not to scale) for SET, RESET, and READ operations on a PCM cell. For phase change material $Ge_2Sb_2Te_5$, $T_{melt} \approx 610°C$ and $T_{cryst} \approx 350°C$.

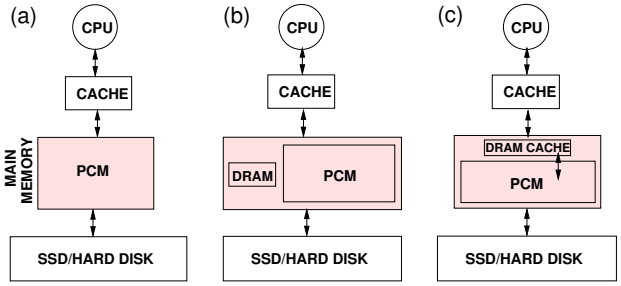


Figure 2: Candidate main memory organizations with PCM.

the primary main memory, and the key challenge of overcoming its write limitations.

2.1 PCM Technology

Phase change memory (PCM) is a byte-addressable non-volatile memory that exploits large resistance contrast between amorphous and crystalline states in so-called phase change materials such as chalcogenide glass. The difference in resistance between the high-resistance amorphous state and the low-resistance crystalline state is typically about five orders of magnitude and can be used to infer logical states of binary data (high represents 0, low represents 1).

Programming a PCM device involves application of electric current, leading to temperature changes that either SET or RESET the cell, as shown schematically in Figure 1. To SET a PCM cell to its low-resistance state, an electrical pulse is applied to heat the cell above the crystalization temperature T_{cryst} (but below the melting temperature T_{melt}) of the phase change material. The pulse is sustained for a sufficiently long period for the cell to transition to the crystalline state. On the other hand, to RESET the cell to its high-resistance amorphous state, a much larger electrical current is applied in order to increase the temperature above T_{melt}. After the cell has melted, the pulse is abruptly cut off, causing the melted material to quench into the amorphous state. To READ the current state of a cell, a small current that does not perturb the cell state is applied to measure the resistance. At normal temperatures ($< 120°C \ll T_{cryst}$), PCM offers many years of data retention.

2.2 Using PCM in the Memory Hierarchy

To see where PCM may fit in the memory hierarchy, we need to know its properties. Table 1 compares PCM with DRAM (technology for today's main memory), NAND flash (technology for today's solid state drives), and HDD (hard disk drives), showing the following points:

- Compared to DRAM, PCM's read latency is close to that of DRAM, while its write latency is about an order of magnitude slower. PCM offers a density advantage over DRAM. This means more memory capacity for the same chip area, or potentially lower price per capacity. PCM is also more energy-efficient than DRAM in idle mode.
- Compared to NAND Flash, PCM can be programmed in place regardless of the initial cell states (i.e., without Flash's expensive "erase" operation). Therefore, its sequential and random accesses show similar (far superior) performance. Moreover, PCM has orders of magnitude higher write endurance than Flash.

Because of these attractive properties, PCM is being incorporated in mobile handsets [24], and recent computer architecture and systems studies have argued that PCM is a promising candidate to be used in main memory in future mainstream computer systems [9, 15, 22].

Figure 2 shows three alternative proposals in recent studies for using PCM in the main memory system [9, 15, 22]. Proposal (a) replaces DRAM with PCM to achieve larger main memory capacity. Even though PCM is slower than DRAM, clever optimizations have been shown to reduce application execution time on PCM to within a factor of 1.2 of that on DRAM [15]. Both proposals (b) and (c) include a small amount of DRAM in addition to PCM so that frequently accessed data can be kept in the DRAM buffer to improve performance and reduce PCM wear. Their difference is that proposal (b) gives software explicit control of the DRAM buffer [9], while proposal (c) manages the DRAM

buffer as another level of transparent hardware cache [22]. It has been shown that a relatively small DRAM buffer (3% size of the PCM storage) can bridge most of the latency gap between DRAM and PCM [22].

2.3 Challenge: Writes to PCM Main Memory

One major challenge in effectively using PCM is overcoming the relative limitations of its write operations. Compared to its read operations, PCM writes incur higher energy consumption, higher latency, lower bandwidth, and limited endurance, as discussed next.

- *High energy consumption:* Compared to reading a PCM cell, a write operation that SETs or RESETs a PCM cell draws higher current, uses higher voltage, and takes longer time (Figure 1). A PCM write often consumes 6–10X more energy than a read [15].
- *High latency and low bandwidth:* In a PCM device, the write latency of a PCM cell is determined by the (longer) SET time, which is about 3X slower than a read operation [15]. Moreover, many PCM prototypes support "iterative writing" of a limited number of bits per iteration in order to limit the instantaneous current level. Several prototypes support $\times 2$, $\times 4$, and $\times 8$ write modes in addition to the fastest $\times 16$ mode [3]. This limitation is likely to hold in the future as well, especially for PCM designed for power-constrained platforms. Because of the limited write bandwidth, writing 64B of data often requires several rounds of writing, leading to the ~ 1 μs write latency in Table 1.
- *Limited endurance:* Existing PCM prototypes have a write endurance ranging from 10^6 to 10^8 writes per cell. With a good wear-leveling algorithm, a PCM main memory can last for several years under realistic workloads [21]. However, because such wear-leveling must be done at the memory controller, the wear-leveling algorithms need to have small memory footprints and be very fast. Therefore, practical algorithms are simple and in many cases, their effectiveness significantly decreases in the presence of extreme hot spots in the memory. For example, even with the wear-leveling algorithms in [21], continuously updating a counter in PCM in a 4GHz machine with 16GB PCM could wear a PCM cell out in about 4 months (without wear-leveling, the cell could wear out in less than a minute).

Recent studies proposed various hardware optimizations to reduce the number of PCM bits written [8, 15, 31, 32]. For example, the PCM controller can perform *data comparison writes*, where a write operation is replaced with a *read-modify-write* operation in order to skip programming unchanged bits [31]. Another proposal is *partial writes* for only dirty words [15]. In both optimizations, when writing a chunk of data in multiple iterations, the set of bits to write in every iteration is often hard-wired for simplicity; if all the hard-wired bits of an iteration are unchanged, the iteration can be skipped [7]. However, these architectural optimizations reduce the volume of writes by only a factor ~ 3. We believe that applications (such as databases) can play an important role in complementing such architectural optimizations by carefully choosing their algorithms and data structures in order to reduce the number of writes, even at the expense of additional reads.

Table 2: General Terms.

Term	Description	Example
E_{rb}	Energy consumed for reading a PCM bit	2 pJ
E_{wb}	Energy consumed for writing a PCM bit	16 pJ
T_l	Latency of accessing a cache line from PCM	230 cycles
T_w	Additional latency of writing a word to PCM	450 cycles
C	size in bytes of the largest CPU cache	8 MB
L	Cache line size in bytes	64B
W	Word size used in PCM writes	8B
N_l	Number of cache line fetches from PCM	-
N_{lw}	Number of cache line write backs to PCM	-
N_w	Number of words written to PCM	-
γ	Avg number of modified bits per modified word	-

3. PCM-FRIENDLY DB ALGORITHMS

In this section, we consider the impact of PCM on database algorithm design. Specifically, reducing PCM writes becomes an important design goal. We discuss general considerations in Section 3.1. Then we re-examine two core database techniques, B^+-tree index and hash joins, in Sections 3.2 and 3.3, respectively.

3.1 Algorithm Design Considerations

Section 2 described three candidate organizations of future PCM main memory, as shown in Figure 2. Their main difference is whether or not to include a transparent or software-controlled DRAM cache. For algorithm design purposes, we consider an abstract framework that captures all three candidate organizations. Namely, we focus on a PCM main memory, and view any additional DRAM as just another (transparent or software-controlled) cache in the hierarchy. This enables us to focus on PCM-specific issues.

Because PCM is the primary main memory, we consider algorithm designs in main memory. There are two traditional design goals for main memory algorithms: (i) low computation complexity, and (ii) good CPU cache performance. Power efficiency has recently emerged as a third design goal. Compared to DRAM, one major challenge in designing PCM-friendly algorithms is to cope with the asymmetry between PCM reads and PCM writes: PCM writes consume much higher energy, take much longer time to complete, and wear out PCM cells (recall Section 2). Therefore, one important design goal of PCM-friendly algorithms is to *minimize PCM writes*.

What granularity of writes should we use in algorithm analysis with PCM main memory: *(a) bits, (b) words, or (c) cache lines*? All three granularities are important for computing PCM metrics. Choice (a) impacts PCM endurance. Choices (a) and (c) affect PCM energy consumption. Choice (b) influences PCM write latency. The drawback of choice (a) is that the relevant metric is the number of *modified* bits (recall that unmodified bits are skipped); this is difficult to estimate because it is often affected not only by the structure of an algorithm, but also by the input data. Fortunately, there is often a simple relationship between (a) and (b). Denote γ as the average number of modified bits per modified word. γ can be estimated for a given input. Therefore, we focus on choices (b) and (c).

Let N_l (T_l) be the number (latency, resp.) of cache line fetches (a.k.a. cache misses) from PCM, N_{lw} be the number of cache line write backs to PCM, and N_w (T_w) be the number (latency, resp.) of modified words written. Let E_{rb} (E_{wb}) be the energy for reading (writing, resp.) a PCM bit. Let L be the number of bytes in a cache line. (Table 2 sum-

marizes the notation used in this paper.) We model the key PCM metrics as follows:

- TotalWear: $NumBitsModified = \gamma N_w$
- $Energy = 8L(N_l + N_{lw})E_{rb} + \gamma N_w E_{wb}$
- $TotalPCMAccessLatency = N_l T_l + N_w T_w$

The total wear and energy computations are straightforward. The latency computation requires explanations. The first part ($N_l T_l$) is the total latency of cache line fetches from PCM. The second part ($N_w T_w$) is the estimated impact of cache line write backs to PCM on the total time. In a traditional system, the cache line write backs are performed asynchronously in the background and often completely hidden. Therefore, algorithm analysis typically ignores the write backs. However, we find that because of the asymmetry of writes and reads, PCM write latency can keep PCM busy for a sufficiently long time to stall front-end cache line fetches significantly. A PCM write consists of (i) a read of the cache line from PCM to identify modified words then (ii) writing modified words in possibly multiple rounds. The above computation includes (ii) as $N_w T_w$, while the latency of (i) ($N_{lw} T_l$) is ignored because it is similar to a traditional cache line write back and thus likely to be hidden.

3.2 B$^+$-Tree Index

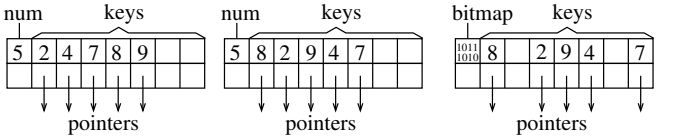
As case studies, we consider two core database techniques for memory-resident data, B$^+$-trees (in the current subsection) and hash joins (in the next subsection), where the main memory is PCM instead of DRAM.

B$^+$-trees are preferred index structures for memory-resident data because they optimize for CPU cache performance. Previous studies recommend that B$^+$-tree nodes be one or a few cache lines large and aligned at cache line boundaries [5, 6, 12, 23]. For DRAM-based main memory, the costs of search/insertion/deletion are similar except in those cases where insertions/deletions incur node splits/merges in the tree. In contrast, for PCM-based main memory, even a normal insertion/deletion that modifies a single leaf node can be more costly than a search in terms of total wear, energy, and elapsed time, because of the writes involved.

We would like to preserve the good CPU cache performance of B$^+$-trees while reducing the number of writes. A cache-friendly B$^+$-tree node is typically implemented as shown in Figure 3(a), where all the keys in the node are sorted and packed, and a counter keeps track of the number of valid keys in the array. The sorted key array is maintained upon insertions and deletions. In this way, binary search can be applied to locate a search key. However, on average, half of the array must be moved to make space for insertions and deletions, resulting in a large number of writes. Suppose that there are K keys and K pointers in the node, and every key, every pointer, and the counter have size equal to the word size W used in PCM writes. Then an insertion/deletion in the sorted node incurs $2(K/2) + 1 = K + 1$ word writes on average.

To reduce writes, we propose two simple unsorted node organizations as shown in Figures 3(b) and 3(c):

- **Unsorted**: As shown in Figure 3(b), the key array is still packed but can be out of order. A search has to scan the array sequentially in order to look for a match or the next smaller/bigger key. On the other hand, an insertion can simply append the new entry to the end of the array, then increment the counter. For a deletion, one can overwrite the entry to delete with the last entry in the array, then decrement the counter. Therefore, an insertion/deletion incurs 3 word writes.
- **Unsorted with bitmap**: We further improve the unsorted organization by allowing the key array to have holes. The counter field is replaced with a bitmap recording valid locations. An insertion writes the new entry to an empty location and updates the bitmap, using 3 word writes, while a deletion updates only the bit in the bitmap, using 1 word write. A search incurs the instruction overhead of a more complicated search process. For 8-byte keys and pointers, a 64-bit bitmap can support nodes up to 1024 bytes large, which is more than enough for supporting typical cache-friendly B$^+$-tree nodes.



(a) Sorted (b) Unsorted (c) Unsorted w/ bitmap

Figure 3: B$^+$-tree node organizations.

Table 3: Terms used in analyzing hash joins.

Term	Description
M_R, M_S	Number of records in relation R and S, respectively
L_R, L_S	Record sizes in relation R and S, respectively
N_{hR}	Number of cache line accesses per hash table visit when building the hash table on R records
N_{hS}	Number of cache line accesses per hash table visit when probing the hash table for S records
$HashTable_{lw}$	Number of line write backs per hash table insertion
$HashTable_w$	Number of words modified per hash table insertion
$MatchPerR$	Number of matches per R record
$MatchPerS$	Number of matches per S record

Given the pros and cons of the three node organizations, we study the following four variants of B$^+$-trees:

- **Sorted:** a normal cache-friendly B$^+$-tree. All the non-leaf and leaf nodes are sorted.
- **Unsorted:** a cache-friendly B$^+$-tree with all the non-leaf and leaf nodes unsorted.
- **Unsorted leaf:** a cache-friendly B$^+$-tree with sorted non-leaf nodes but unsorted leaf nodes. Because most insertions/deletions do not modify non-leaf nodes, the unsorted leaf nodes may capture most of the benefits.
- **Unsorted leaf with bitmap:** This variant is the same as unsorted leaf except that leaf nodes are organized as unsorted nodes with bitmaps.

Our experimental results in Section 4 show that the unsorted schemes can significantly improve total wear, energy consumption, and run time for insertions and deletions. Among the three unsorted schemes, unsorted leaf is the best for index insertions and it incurs negligible index search overhead, while unsorted leaf with bitmap achieves the best index deletion performance.

3.3 Hash Joins

One of the most efficient join algorithms, hash joins are widely used in data management systems. Several cache-friendly variants of hash joins are proposed in the literature [1, 4, 27]. Most of these algorithms are based on the

Algorithm 1 Existing algorithm: simple hash join.

Build phase:

```
1: for (i = 0; i < M_R; i++) do
2:    r= record i in Relation R;
3:    insert r into hash table;
```

Probe phase:

```
1: for (j = 0; j < M_S; j++) do
2:    s= record j in Relation S;
3:    probe s in the hash table;
4:    if there are match(es) then
5:       generate join result(s) from the matching records;
6:       send join result(s) to the upper-level operator;
```

Algorithm 2 Existing algorithm: cache partitioning.[2]

Partition phase:

```
1: htsize = M_R* hash_table_per_entry_metadata_size;
2: P = ⌈(M_R L_R + M_S L_S + htsize)/C⌉;
3: for (i = 0; i < M_R; i++) do {partition R}
4:    r= record i in Relation R;
5:    p= hash(r) modulo P;
6:    copy r to partition R_p;
7: for (j = 0; j < M_S; j++) do {partition S}
8:    s= record j in Relation S;
9:    p= hash(s) modulo P;
10:   copy s to partition S_p;
```

Join phase:

```
1: for (p = 0; p < P; p++) do
2:    join R_p and S_p using simple hash join;
```

following two representative algorithms. (Table 3 defines the terms used in describing and analyzing the algorithms.)

Simple Hash Join. As shown in Algorithm 1, in the build phase, the algorithm scans the smaller build relation R. For every build record, it computes a hash code from the join key, and inserts the record into the hash table. In the probe phase, the algorithm scans the larger probe relation S. For every probe record, it computes the hash code, and probes the hash table. If there are matching build records, the algorithm computes the join results, and sends them to upper level query operators.

The cost of this algorithm can be analyzed as in Table 4 with the terms defined in Table 3. Here, we assume that the hash table does not fit into CPU cache, which is usually the case. We do not include PCM costs for the join results as they are often consumed in the CPU cache by higher-level operators in the query plan tree.

The cache misses of the build phase are caused by reading all the join keys ($\min(\frac{M_R L_R}{L}, M_R)$) and accessing the hash table ($M_R N_{hR}$). When the record size is small, the first term is similar to reading the entire build relation. When the record size is large, it incurs roughly one cache miss per record. Note that because multiple entries may share a single hash bucket, the lines written back can be a subset of the lines accessed for a hash table visit. For the probe phase, the cache misses are caused by scanning the probe relation ($\frac{M_S L_S}{L}$), accessing the hash table ($M_S N_{hS}$), and accessing matching build records in a random fashion. The latter can be computed as shown in Figure 4. The other computations are straightforward.

Algorithm 3 Our proposal: virtual partitioning.[2]

Partition phase:

```
1: htsize = M_R* hash_table_per_entry_metadata_size;
2: P = ⌈((M_R + M_S)2 + M_R(L_R − 1 + L) + M_S(L_S − 1 +
   L) + htsize)/C⌉;
3: initiate ID lists RList[0..P − 1] and SList[0..P − 1];
4: for (i = 0; i < M_R; i++) do {virtually partition R}
5:    r= record i in Relation R;
6:    p= hash(r) modulo P;
7:    append ID i into RList[p];
8: for (j = 0; j < M_S; j++) do {virtually partition S}
9:    s= record j in Relation S;
10:   p= hash(s) modulo P;
11:   append ID j into SList[p];
```

Join phase:

```
1: for (p = 0; p < P; p++) do {join R_p and S_p}
2:    for each i in RList[p] do
3:       r= record i in Relation R;
4:       insert r into hash table;
5:    for each j in SList[p] do
6:       s= record j in Relation S;
7:       probe s in the hash table;
8:       if there are match(es) then
9:          generate join result(s) from the matching records;
10:         send join result(s) to the upper-level operator;
```

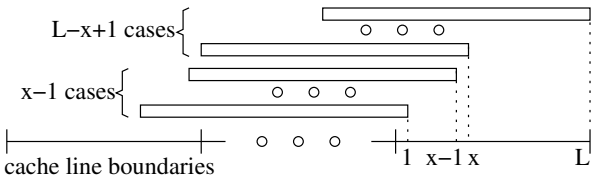


Figure 4: Computing average number of cache misses for unaligned records. A record of $size = yL+x$ bytes, $y \geq 0, L > x \geq 0$, has L possible locations relative to cache line boundaries. Accessing the record incurs on average $\frac{x-1}{L}2 + \frac{L-x+1}{L} + y = \frac{size-1}{L} + 1$ cache misses.

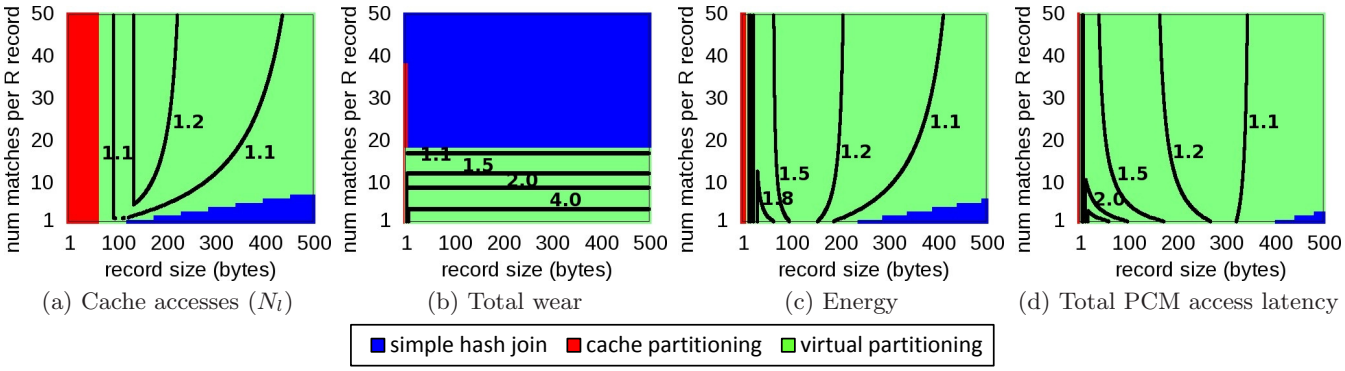
Cache Partitioning. When both input relation sizes are fixed, if we reduce the record sizes (L_R, L_S), then the numbers of records (M_R, M_S) increase. Therefore, simple hash join incurs a large number of cache misses when record sizes are small. The cache partitioning algorithm solves this problem. As shown in Algorithm 2, in the partition phase, the two input relations (R and S) are hash partitioned so that every pair of partitions (R_p and S_p) can fit into the CPU cache. Then in the join phase, every pair of R_p and S_p are joined using the simple hash join algorithm.

The cost analysis of cache partitioning is straightforward as shown in Table 4. Note that we assume that modified cache lines during the partition phase are not prematurely evicted because of cache conflicts. Observe that the number of cache misses using cache partitioning is constant if the relation sizes are fixed. This addresses the above problem of simple hash join.

[2]For simplicity, Algorithm 2 and Algorithm 3 assume perfect partitioning when generating cache-sized partitions. To cope with data skews, one can increase the number of partitions P so that even the largest partition can fit into the CPU cache. Note that using a larger P does not change the algorithm analysis.

Table 4: Cost analysis for three hash join algorithms.

Algorithm		Cache Line Accesses from PCM (N_l)	Cache Line Write Backs (N_{lw})	Words Written (N_w)
Simple Hash	Build	$\min(\frac{M_R L_R}{L}, M_R) + M_R N_{hR}$	$M_R HashTable_{lw}$	$M_R HashTable_w$
	Probe	$\frac{M_S L_S}{L} + M_S N_{hS} + M_S MatchPerS(\frac{L_R - 1}{L} + 1)$	0	0
Cache Partition	Partition	$2(\frac{M_R L_R}{L} + \frac{M_S L_S}{L})$	$\frac{M_R L_R}{L} + \frac{M_S L_S}{L}$	$\frac{M_R L_R}{W} + \frac{M_S L_S}{W}$
	Join	$\frac{M_R L_R}{L} + \frac{M_S L_S}{L}$	0	0
Virtual Partition	Partition	$\frac{M_R L_R}{L} + \frac{M_S L_S}{L} + (M_R + M_S)\frac{2}{L}$	$(M_R + M_S)\frac{2}{L}$	$(M_R + M_S)\frac{2}{W}$
	Join	$(M_R + M_S)\frac{2}{L} + M_R(\frac{L_R - 1}{L} + 1) + M_S(\frac{L_S - 1}{L} + 1)$	0	0



(a) Cache accesses (N_l) (b) Total wear (c) Energy (d) Total PCM access latency

Figure 5: Comparing three hash join algorithms analytically. ($L_S = L_R$, $MatchPerS = 1$, $\gamma = 0.5$; the hash table in simple hash join does not fit into cache; hash table access parameters are based on experimental measurements: $N_{hR} \simeq N_{hS} = 1.8$, $HashTable_{lw} = 1.5$, $HashTable_w = 5.0$.) For configurations where virtual partitioning is the best scheme, contour lines show the relative benefits of virtual partitioning compared to the second best scheme.

However, cache partitioning introduces a large number of writes compared to simple hash join: it is writing the amount of data equivalent to the size of the entire input relations. As writes are bad for PCM, we would like to design an algorithm that reduces the writes while still achieving similar benefits of cache partitioning. We propose the following variant of cache partitioning.

New: Virtual Partitioning. Instead of physically copying input records into partitions, we perform the partitioning *virtually*. As shown in Algorithm 3, in the partition phase, for every partition, we compute and remember the record IDs that belong to the partition for both R and S.[3] Then in the join phase, we can use the record ID lists to join the records of a pair of partitions in place, thus avoiding the large number of writes in cache partitioning.

We optimize the record ID list implementation by storing the deltas of two subsequent record IDs to further reduce the writes. As the number of cache partitions is often smaller than a thousand, we find using two-byte integers can encode most deltas. For rare cases with larger deltas, we reserve $0xFFFF$ to indicate that a full record ID is recorded next.

The costs of the virtual partitioning algorithm is analyzed in Table 4. The costs for the partition phase include scanning the two relations as well as generating the record ID lists. The latter writes two bytes per record. In the join phase, the records are accessed in place. They are essentially scattered in the two input relations. Therefore, we use the formula for unaligned records in Figure 4 to compute the number of cache misses for accessing the build and probe records. Note that the computation of the number of partitions P in Algorithm 3 guarantees that the total cache lines accessed per pair of R_p and S_p fit into the CPU cache capacity C.

Comparisons of the Three Algorithms. Figure 5 compares the three algorithms analytically using the formulas in Table 4. We assume R and S have the same record size, and it is a primary-foreign key join (thus $MatchPerS = 1$). From left to right, Figures 5(a) to (d) show the comparison results for four metrics: (a) cache accesses (N_l), (b) total wear, (c) energy, and (d) total PCM access latency. In each figure, we vary the record size from 1 to 500 bytes, and the number of matches per R record ($MatchPerR$) from 1 to 50. Every point represents a configuration for the hash join. The color of a point shows the best scheme for the corresponding configuration: blue for simple hash join, red for cache partitioning, and green for virtual partitioning. For configurations where virtual partitioning is the best scheme, the contour lines show the relative benefits of virtual partitioning compared to the second best scheme.

Figure 5(a) focuses on CPU cache performance, which is the main consideration for previous cache-friendly hash join designs. We see that as expected, simple hash join is the best scheme when record size is very large and cache partitioning is the best scheme when record size is small. Compared to simple hash join, virtual partitioning avoids the many cache misses caused by hash table accesses. Compared to

[3]We assume that there is a simple mapping between a record ID and the record location in memory. For example, if fixed length records are stored consecutively in an array, then the array index can be used as the record ID. If records always start at 8B boundaries, then the record ID of a record can be the record starting address divided by 8.

cache partitioning, virtual partitioning reduces the number of cache misses during the partition phase, while paying extra cache misses for accessing scattered records in the join phase. As a result, virtual partitioning achieves the smallest number of cache misses for a large number of configurations in the middle between the red and blue points.

Figures 5(b) to (d) show the comparison results for the three PCM metrics. First of all, we see that the figures are significantly different from Figure 5(a). This means that introducing PCM main memory can significantly impact the relative benefits of the algorithms. Second, very few configurations benefit from cache partitioning because it incurs a large number of PCM writes in the partition phase, adversely impacting its PCM performance. Third, in Figure 5(b), virtual partitioning achieves the smallest number of writes when $MatchPerR \leq 18$. Virtual partitioning avoids many of the expensive PCM writes in the partition phase of the cache partitioning algorithm. Interestingly, simple hash join achieves the smallest number of writes when $MatchPerR \geq 19$. This is because as $MatchPerR$ increases, the number of S records (M_S) increases proportionally, leading to a larger number of PCM writes for virtual partitioning, while the number of PCM writes in simple hash join is not affected. The cross-over point is 19 here. Finally, virtual partitioning presents a good balance between cache line accesses and PCM writes, and it excels in energy and total PCM access latency in most cases.

4. EXPERIMENTAL EVALUATION

We evaluate our proposed B$^+$-tree and hash join algorithms through cycle-accurate simulations in this section. We start by describing the simulator used in the experiments. Then we present the experimental results for B$^+$-trees and hash joins. Finally, we perform sensitivity analysis for PCM parameters.

4.1 Simulation Platform

We extended a cycle-accurate out-of-order X86-64 simulator, PTLsim [20], with PCM support. PTLsim is used extensively in computer architecture studies and is currently the only publicly available cycle-accurate simulator for out-of-order x86 micro-architectures. The simulator models the details of a superscalar out-of-order processor, including instruction decoding, micro-code, branch prediction, function units, speculation, and a three-level cache hierarchy. PTLsim has multiple use modes; we use PTLsim to simulate single 64-bit user-space applications in our experiments.

We extended PTLsim in the following ways to model PCM. First, we model data comparison writes for PCM writes. When writing a cache line to PCM, we compare the new line with the original line to compute the number of modified bits and the number of modified words. The former is used to compute PCM energy consumption, while the latter impacts PCM write latency. Second, we model four parallel PCM memory ranks. Accesses to different ranks can be carried out in parallel. Third, we model the details of cache line write back operations carefully. Previously, PTLsim assumes that cache line write backs can be hidden completely, and does not model the details of this operation. Because PCM write latency is significantly longer than its read latency, cache line write backs may actually keep the PCM busy for a sufficiently long time to stall front-end cache line fetches. Therefore, we implemented a 32-entry FIFO write queue in the on-chip memory controller, which keeps track of dirty cache line evictions and performs the PCM writes asynchronously in the background.

Table 5: Simulation Setup.

Simulator	PTLsim enhanced with PCM support
Processor	Out-of-order X86-64 core, 3GHz
CPU cache	Private L1D (32KB, 8-way, 4-cycle latency), private L2 (256KB, 8-way, 11-cycle latency), shared L3 (8MB, 16-way, 39-cycle latency), all caches with 64B lines, 64-entry DTLB, 32-entry write back queue
PCM	4 ranks, read latency for a cache line: 230 cycles, write latency per 8B modified word: 450 cycles, E_{rb} = 2 pJ, E_{wb} = 16 pJ

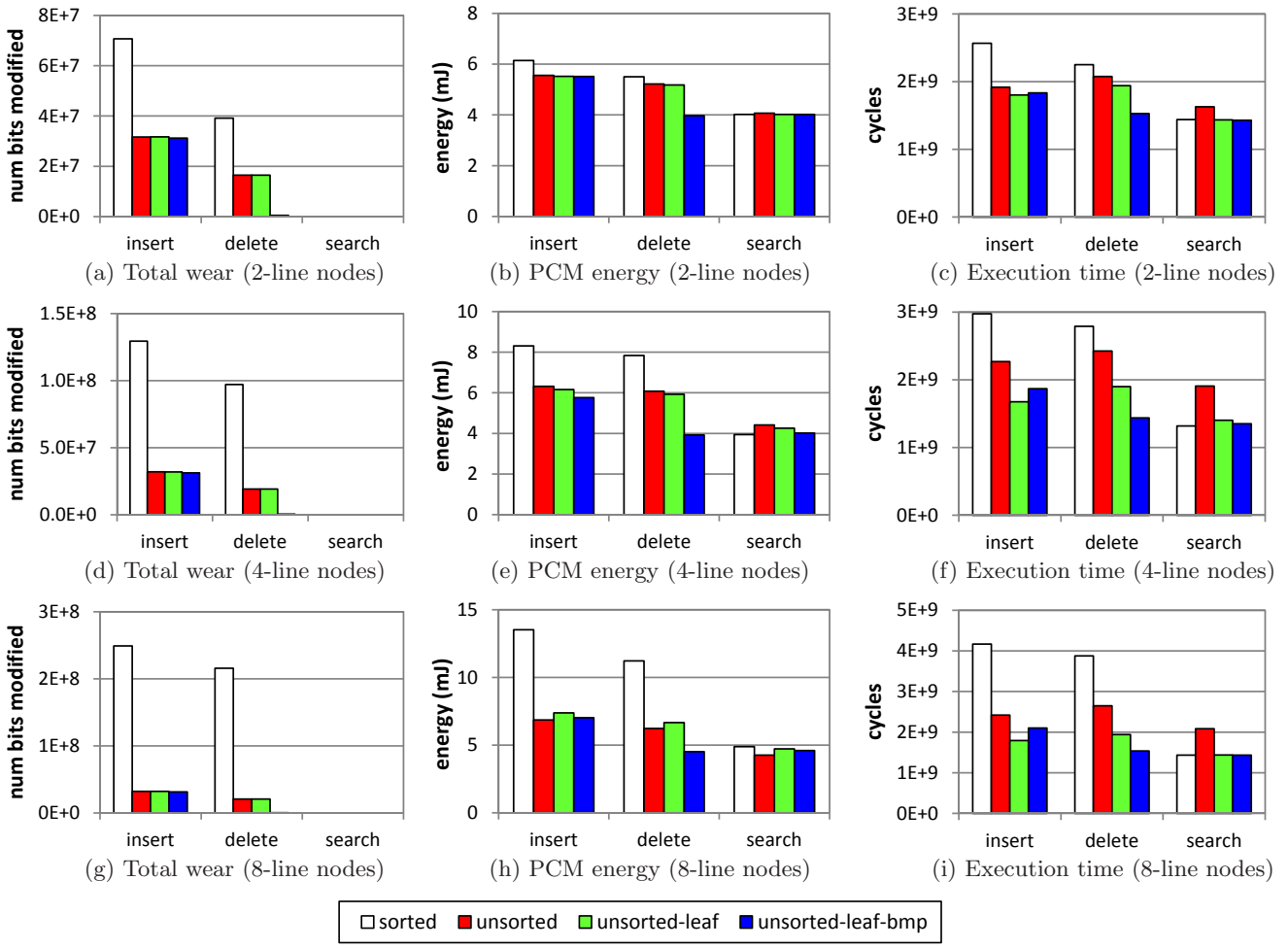
Table 5 describes the simulation parameters. The cache hierarchy is modeled after the recent Intel Nehalem processors [14]. The PCM latency and energy parameters are based on a previous computer architecture study [15]. We adjusted the latency in cycles according to the 3GHz processor frequency and the DDR3 bus latency. The word size of 8 bytes per iteration of write operations is based on [8].

4.2 B$^+$-Tree Index

We implemented four variants of B$^+$-trees as described in Section 3.2: sorted, unsorted, unsorted leaf, and unsorted leaf with bitmap. Figure 6 compares the four schemes for common index operations. In every experiment, we populate the trees with 50 million entries. An entry consists of an 8-byte integer key and an 8-byte pointer. We populate the nodes 75% full initially. We randomly shuffle the entries in all unsorted nodes so that the nodes represent the stable situations after updates. Note that the total tree size is over 1GB, much larger than the largest CPU cache (8MB). For the insertion experiments, we insert 500 thousand random new entries into the trees back to back, and report total wear in number of PCM bits modified, PCM energy consumption in millijoules, and execution time in cycles for the entire operation. Similarly, we measure the performance of 500 thousand back-to-back deletions for the deletion experiments, and 500 thousand back-to-back searches for the search experiments. We vary the node size of the trees. As suggested by previous studies, the best tree node sizes are a few cache lines large [5, 12]. Since a one-line (64B) node can contain only 3 entries, which makes the tree very deep, we show results for node sizes of 2, 4, and 8 cache lines.

The sub-figures in Figure 6 are arranged as a 3x3 matrix. Every row corresponds to a node size. Every column corresponds to a performance metric. In every sub-figure, there are three groups of bars, corresponding to the insertion, deletion, and search experiments. The bars in each group show the performance of the four schemes. (Note that search does not incur any wear.) We observe the following points in Figure 6.

First, compared to the conventional sorted trees, all the three unsorted schemes achieve better total wear, energy consumption, and execution time for insertions and deletions, the two index operations that incur PCM writes. The sorted trees pay the cost of moving the sorted array of entries in a node to accommodate insertions and deletions. In contrast, the unsorted schemes all save PCM writes by allowing entries to be unsorted upon insertions and deletions. This saving increases as the node size increases. Therefore,



(a) Total wear (2-line nodes) (b) PCM energy (2-line nodes) (c) Execution time (2-line nodes)

(d) Total wear (4-line nodes) (e) PCM energy (4-line nodes) (f) Execution time (4-line nodes)

(g) Total wear (8-line nodes) (h) PCM energy (8-line nodes) (i) Execution time (8-line nodes)

Figure 6: B$^+$-tree performance. (50 million entries in trees; 75% full; "insert": inserting 500 thousand random keys; "delete": randomly deleting 500 thousand existing keys; "search": searching for 500 thousand random keys)

the performance gaps widen as the node size grows from 2 cache lines to 8 cache lines.

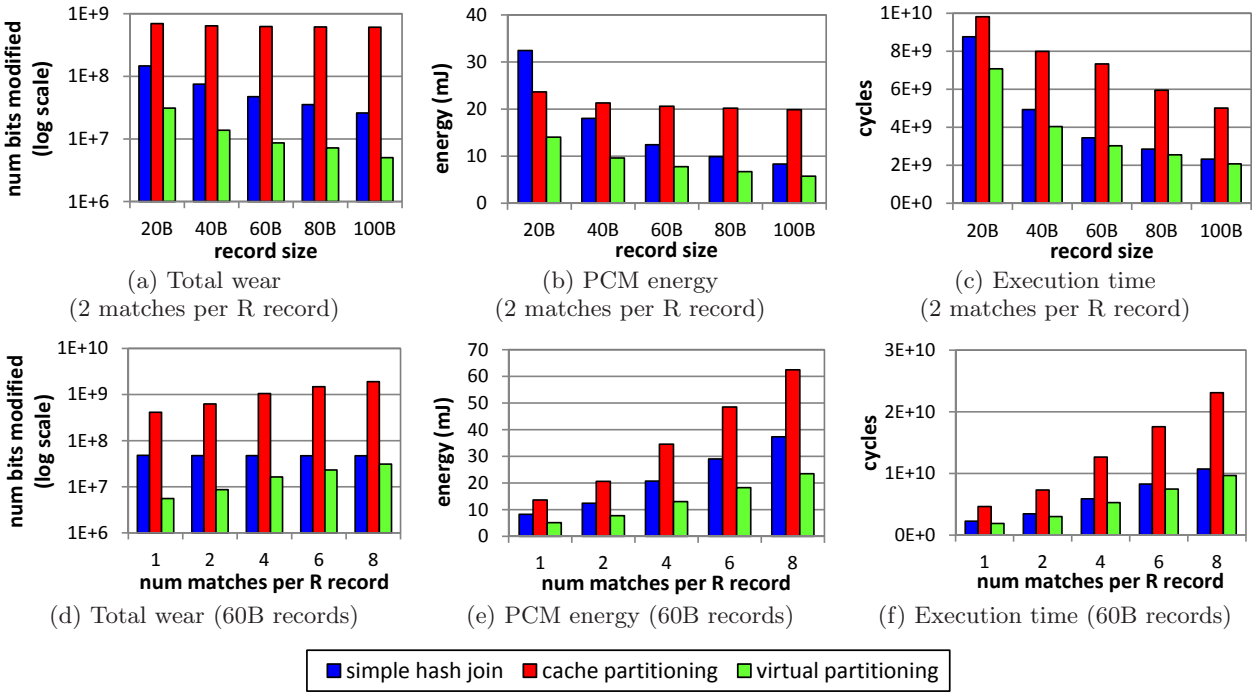
Second, compared to the conventional sorted trees, the scheme with all nodes unsorted suffers from slower search time by a factor of 1.13–1.46X because the hot, top tree nodes stay in CPU cache, and a search incurs a lot of instruction overhead in the unsorted non-leaf nodes. In contrast, the two schemes with only unsorted leaf nodes achieve similar search time as the sorted scheme.

Third, comparing the two unsorted leaf schemes, we see that unsorted leaf with bitmap achieves better total wear, energy, and time for deletions. This is because unsorted leaf with bitmap often only needs to mark one bit in a leaf bitmap for a deletion (and the total wear is about $5E5$ bits modified), while unsorted leaf has to overwrite the deleted entry with the last entry in a leaf node and update the counter in the node. On the other hand, the unsorted leaf with bitmap suffers from slightly higher insertion time because of the instruction overhead of handling the bitmap and the holes in a leaf node.

Overall, we find that the two unsorted leaf schemes achieve the best performance. Compared to the conventional sorted B$^+$-tree, the unsorted leaf schemes improve total wear by a factor of 7.7–436X, energy consumption by a factor of 1.7–2.5X, and execution time by a factor of 2.0–2.5X for insertions and deletions, while achieving similar search performance. If the index workload consists of mainly insertions and searches (with the tree size growing), we recommend the normal unsorted leaf. If the index workload contains a lot of insertions and a lot of deletions (e.g., the tree size stays roughly the same), we recommend the unsorted leaf scheme with bitmap.

4.3 Hash Joins

We implemented the three hash join algorithms as discussed in Section 3.3: simple hash join, cache partitioning, and virtual partitioning. We model in-memory join operations, where the input relations R and S are in main memory. The algorithms build in-memory hash tables on the R relation. To hash a R record, we compute an integer hash code from its join key field, and modulo this hash code by the size of the hash table to obtain the hash slot. Then we insert (hash code, pointer to the R record) into the hash slot. Conflicts are resolved through chained hashing. To probe an S record, we compute the hash code from its join key field, and use the hash code to look up the hash ta-



(a) Total wear (2 matches per R record)
(b) PCM energy (2 matches per R record)
(c) Execution time (2 matches per R record)
(d) Total wear (60B records)
(e) PCM energy (60B records)
(f) Execution time (60B records)

Figure 7: Hash join performance. (50MB R table joins S table, varying the record size from 20B to 100B and varying the number of matches per R record from 1 to 8.)

ble. When there is an entry with the matching hash code, we check the associated R record to make sure that the join keys actually match. The join results are sent to a high-level operator that consumes the results. In our implementation, the high-level operator simply increments a counter.

Figure 7 compares the three hash join algorithms. The R relation is 50MB large. Both relations have the same record size. We vary the record size from 20B to 100B in Figures 7(a)–(c). We vary the number of matches per R record ($MatchPerR$) from 1 to 8 in Figures 7(d)–(f); in other words, the size of S varies from 50MB to 400MB. We report total wear, energy consumption, and execution times for every set of experiments.

The results in Figure 7 confirm our analytical comparison in Section 3.3. First, cache partitioning performs poorly in almost all cases because it performs a large number of PCM writes in its partition phase. This results in much higher total wear, higher energy consumption, and longer execution time compared to the other two schemes.

Second, compared to simple hash join, when varying record size from 20B to 100B, virtual partitioning improves total wear by a factor of 4.7–5.2X, energy consumption by a factor of 2.3–1.4X, and execution time by a factor of 1.24–1.12X. When varying $MatchPerR$ from 1 to 8, virtual partitioning improves total wear by a factor of 8.6–1.5X, energy consumption by a factor of 1.61–1.59X, and execution time by a factor of 1.19–1.11X.

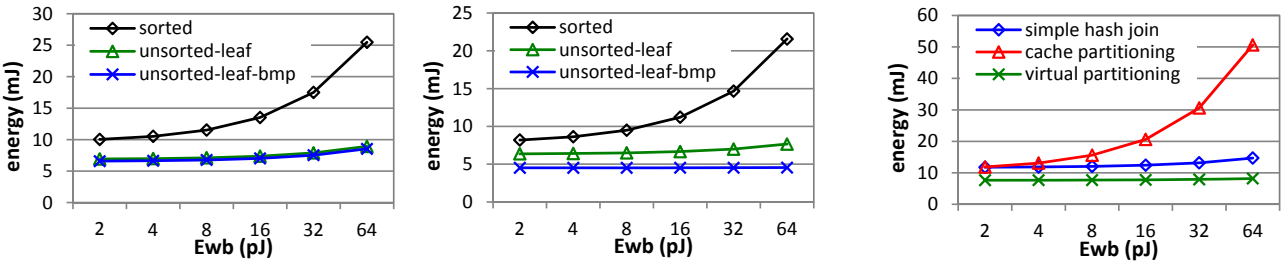
Overall, virtual partitioning achieves the best performance among the three schemes in all the experiments. Compared to cache partitioning, virtual partitioning avoids copying data in the partition phase by remembering record IDs per partition. Compared to simple hash join, virtual partitioning avoids excessive cache misses due to hash table accesses. Therefore, virtual partitioning achieves good behaviors for both PCM writes and cache accesses. Note that the record size and $MatchPerR$ settings in the experiments fall in the region where virtual partitioning wins in Figure 5. Therefore, the experimental results confirm our analytical comparison in Section 3.3.

4.4 PCM Parameter Sensitivity Analysis

In this section, we vary the energy and latency parameters of PCM in the simulator, and study the impact of the parameter changes on the performance of the B$^+$-tree and hash join algorithms. Note that we still assume data comparison writes for PCM write.

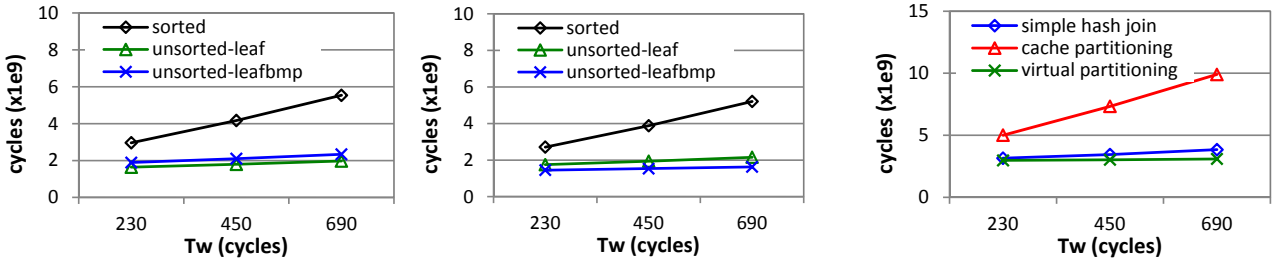
Figure 8 varies the energy consumed by writing a PCM bit (E_{wb}) from 2pJ to 64pJ. The default value of E_{wb} is 16pJ, and 2pJ is the same as the energy consumed by reading a PCM bit. From left to right, Figures 8(a)–(c) show the impact of varying E_{wb} on the energy consumptions of B$^+$-tree insertions, B$^+$-tree deletions, and hash joins. First, we see that as E_{wb} gets smaller, the curves become flat; the energy consumption is more and more dominated by the cache line fetches for reads and for data comparison writes. Second, as E_{wb} gets larger, the curves increase upwards because the larger E_{wb} contributes significantly to the overall energy consumption. Third, changing E_{wb} does not qualitatively change our previous conclusions. For B$^+$-trees, the two unsorted leaf schemes are still better than sorted B$^+$-trees. Among the three hash join algorithms, virtual partitioning is still the best.

Figure 9 varies the latency of writing a word to PCM (T_w) from 230 cycles to 690 cycles. The default T_w is 450 cycles, and 230 is the same latency as reading a cache line from PCM. From left to right, Figures 8(a)–(c) show the impact of varying T_w on the execution times of B$^+$-tree insertions, B$^+$-tree deletions, and hash joins. We see that as T_w increases, the performance gaps among different schemes become larger. (The performance gap between simple hash



(a) B$^+$-tree insertions (b) B$^+$-tree deletions (c) Hash joins (2 matches per R record, 60B)

Figure 8: Sensitivity analysis: varying energy consumed for writing a PCM bit (E_{wb}).



(a) B$^+$-tree insertions (b) B$^+$-tree deletions (c) Hash joins (2 matches per R record, 60B)

Figure 9: Sensitivity analysis: varying latency of writing a word to PCM (T_w).

join and virtual partitioning is 6% when T_w is 230 cycles.) We find that previous conclusions still hold for B$^+$-trees and hash joins.

5. RELATED WORK

PCM Architecture. As discussed in previous sections, several recent studies from the computer architecture community have proposed solutions to make PCM a replacement for or an addition to DRAM main memory. These studies address various issues including improving endurance [15, 21, 22, 32], improving write latency by reducing the number of PCM bits written [8, 15, 31, 32], preventing malicious wear-outs [26], and supporting error corrections [25]. However, these studies focus on hardware design issues that are orthogonal to our focus on designing efficient algorithms for software running on PCM.

PCM-Based File Systems. BPFS [9], a file system designed for byte-addressable persistent memory, exploits both the byte-addressability and non-volatility of PCM. In addition to being significantly faster than disk-based file systems (even when they are run on DRAM), BPFS provides strong safety and consistency guarantees by using a new technique called *short-circuit shadow paging*. Unlike traditional shadow paging file systems, BPFS uses copy-on-write at fine granularity to atomically commit small changes at any level of the file system tree. This avoids updates to the file system triggering a cascade of copy-on-write operations from the modified location up to the root of the file system tree. BPFS is a file system, and hence it does not consider the database algorithms we consider. Moreover, BPFS has been designed for the general class of byte-addressable persistent memory, and it does not consider PCM-specific issues such as read-write asymmetry or limited endurance.

Battery-Backed DRAM. Battery-backed DRAM (BBDRAM) has been studied as a byte-addressable, persistent memory. The Rio file cache [16] uses BBDRAM as the buffer cache, eliminating any need to flush dirty data to disk. The Rio cache has also been integrated into databases as a persistent database buffer cache [18]. The Conquest file system [29] uses BBDRAM to store small files and metadata. eNVy [30] placed flash memory on the memory bus by using a special controller equipped with a BBDRAM buffer to hide the block-addressable nature of flash. WAFL [13] keeps file system changes in a log in BBDRAM and only occasionally flushes them to disk. While BBDRAM may be an alternative to PCM, PCM has two main advantages over BBDRAM. First, BBDRAM is vulnerable to correlated failures; for example, the UPS battery will often fail either before or along with primary power, leaving no time to copy data out of DRAM. Second, PCM is expected to scale much better that DRAM, making it a better long-term option for persistent storage [3]. On the other hand, using PCM requires dealing with expensive writes and limited endurance, a challenge not present with BBDRAM. Therefore, BBDRAM-based algorithms do not require addressing the challenges studied in this paper.

Main Memory Database Systems and Cache-Friendly Algorithms. Main memory database systems [11] maintain necessary data structures in DRAM and hence can exploit DRAM's byte-addressable property. As discussed in Section 3.1, the traditional design goals of main memory algorithms are low computation complexity and good CPU cache performance. Like BBDRAM-based systems, main memory database systems do not need to address PCM-specific challenges. In this paper, we found that for PCM-friendly algorithms, one important design goal is to minimize PCM writes. Compared to previous cache-friendly B$^+$-trees and hash joins, our new algorithms achieve significantly better performance in terms of PCM total wear, energy consumption, and execution time.

6. CONCLUSION

A promising non-volatile memory technology, PCM is expected to play an important role in the memory hierarchy in the near future. This paper focuses on exploiting PCM

as main memory for database systems. Based on the unique characteristics of PCM (as opposed to DRAM and NAND flash), we identified the importance of reducing PCM writes for optimizing PCM endurance, energy, and performance. Specifically, we applied this observation to database algorithm design, and proposed new B^+-tree and hash join designs that significantly improve the state-of-the-art.

As future work in the PCM-DB project, we are interested in optimizing PCM writes for different aspects of database system designs, including important data structures, query processing algorithms, and transaction logging and recovery. The latter is important for achieving transaction atomicity and durability. BPFS proposed a different solution based on shadow copying and atomic writes [9]. It is interesting to compare this proposal with conventional database transaction logging, given the goal of reducing PCM writes.

Moreover, another interesting aspect to study is the fine-grain non-volatility of PCM. Challenges may arise in hierarchies where DRAM is explicitly controlled by software. Because DRAM contents are lost upon restart, the relationship between DRAM and PCM must be managed carefully; for example, pointers to DRAM objects should not be stored in PCM. On the other hand, the fine-grain non-volatility may enable new features, such as "instant-reboot" that resumes the execution states of long-running queries upon crash recovery so that useful work is not lost.

7. REFERENCES

[1] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[2] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[3] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy. Phase change memory technology. *J. Vacuum Science*, 28(2), 2010.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, 2004.

[5] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD*, 2001.

[6] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B^+-trees: Optimizing both cache and disk performance. In *SIGMOD*, 2002.

[7] S. Cho. Personal communication, 2010.

[8] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO*, 2009.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[10] E. Doller. Phase change memory and its impacts on memory hierarchy. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[11] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE TKDE*, 4(6), 1992.

[12] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B^+-trees. In *SIGMETRICS*, 2003.

[13] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994.

[14] Intel Corp. First the tick, now the tock: Intel micro-architecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[16] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. *Operating Systems Review*, 31, 1997.

[17] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *The VLDB Journal*, 19(1), 2010.

[18] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. *The VLDB Journal*, 7(3), 1998.

[19] PCM-DB. http://www.pittsburgh.intel-research.net/projects/hi-spade/pcm-db/.

[20] PTLsim. http://www.ptlsim.org/.

[21] M. K. Qureshi, J. P. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO*, 2009.

[22] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[23] J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In *SIGMOD*, 2000.

[24] Samsung. Samsung ships industry's first multi-chip package with a PRAM chip for handsets. http://www.samsung.com/us/business/semiconductor/newsView.do?news_id=1149, April 2010.

[25] S. E. Schechter, G. H. Loh, K. Straus, and D. Burger. Use ECP, not ECC, for hard failures in resistive memories. In *ISCA*, 2010.

[26] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *ISCA*, 2010.

[27] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, 1994.

[28] H.-W. Tseng, H.-L. Li, and C.-L. Yang. An energy-efficient virtual memory system with flash memory as the secondary storage. In *Int'l Symp. on Low Power Electronics and Design (ISPLED)*, 2006.

[29] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *USENIX Annual Technical Conference*, 2002.

[30] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *ASPLOS*, 1994.

[31] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *IEEE ISCAS*, 2007.

[32] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.

SwissBox: An Architecture for Data Processing Appliances

G. Alonso D. Kossmann T. Roscoe
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
www.systems.ethz.ch

ABSTRACT

Database appliances offer fully integrated hardware, storage, operating system, database, and related software in a single package. Database appliances have a relatively long history but the advent of multicore architectures and cloud computing have facilitated and accelerated the demand for such integrated solutions. Database appliances represent a significant departure from the architecture of traditional systems mainly because of the cross layer optimizations that are possible. In this paper we describe *SwissBox*, an architecture for data processing appliances being developed at the Systems Group of ETH Zurich. SwissBox combines a number of innovations at all system levels – from customized hardware (FPGAs), an operating systems that treats multicore machines as distributed systems (Barrelfish), to an elastic storage manager that takes advantage of both muticores and clusters (Crescando)– to provide a completely new platform for system development and database research. In the paper we motivate the architecture with several use cases and discuss each one of its components in detail, pointing out the challenges to solve and the advantages that cross-layer optimizations can bring in practice.

1. INTRODUCTION

Database appliances optimize the design, operation, and maintenance costs of data processing solutions by integrating all the components into a single system. Thus, an appliance typically uses customized hardware and storage, an operating system tailored to the application, and software highly optimized to the underlying platform. The advantage of such an approach is that cross-layer optimization and a high degree of customization can significantly improve the behavior and characteristics of the system over what is possible using independent, off-the-shelf components.

The recent interest in appliances is the result of a combination of factors. First, applications such as cloud computing and social networks have raised the bar for performance and scalability requirements. Second, existing cluster-based solutions implementing multi-tier architectures have proven to be both costly and difficult to maintain, opening the door for tightly integrated, closed systems maintained by the vendor. Third, the advent of multicore and fast networks allows enormous processing capacity to be packaged in just such a box. Last but not least, the enormous cost of today's infrastructure gives users plenty of motivation to consider customized solutions rather than using general purpose systems, creating the opportunity for highly tailored approaches focused on narrow vertical markets, or even on single use cases in large enough systems.

These trends, combined with the requirements from a number of real use cases, are the motivation for *SwissBox*: an architecture for data processing appliances we are developing in the Systems Group at ETH Zurich. SwissBox has two broad goals. First, we are addressing the challenges in engineering new appliances in the face of diverse and rapidly changing hardware and workloads.

Second, we are exploring the full ramifications of the freedom granted by the appliance model to re-architect the entire data processing stack from the hardware up to the application.

1.1 Background: Database appliances

The concept of a database appliance, broadly understood, has a long history. There are, however, important differences between current and past approaches to building appliances. Today's appliances use common off-the-shelf components (blade servers, operating systems, database engines, etc.) combined into a customized architecture. In contrast, previous, appliances like the *Gamma* database machine [4] were built using specialized hardware, with the consequence that their performance could be soon outstripped by rapidly advancing commodity systems. To illustrate this point, we briefly discuss three representative modern appliances: SAP's *Business Warehouse Accelerator*, the Netezza *TwinFin*, and Oracle's *Exadata*.

SAP's Business Warehouse Accelerator (BWA) speeds up access to a data warehouse by exploiting column stores, indexes in main memory, and multicore data processing. It consists of an array of computing blades plus storage nodes connected through a high-speed network switch. The data is read from the original database and reorganized into a star schema, reformatted as a column store, indexed, compressed, and written to the storage nodes. Query processing is done in the main memory of independent cores using on-the-fly aggregation, with the indexes also kept in main memory.

Oracle's Exadata combines "intelligent storage nodes" and database processing nodes in a shared disk configuration (based on Oracle RAC) connected by Infiniband. Exadata pushes relational operators like projections and join filtering to the storage nodes, thus reducing I/O overhead and network traffic. This is implemented with a function shipping module that complements the data shipping capability of the (conventional) database engine. To further reduce access latency, Exadata uses a Flash-based cache between the storage and the database nodes for hotspot data.

Netezza's TwinFin is based on a grid of "S-Blades" with a redundant front-end node. The grid is based on a custom IP-based

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

network fabric. Each S-Blade is a multicore processor where each core has an associated FPGA and disk drive. Data is streamed from the disks to the cores, with the FPGAs acting as filters which decompress the data and also execute relational operators such as selection and projection. In this way, an important part of the SQL processing is offloaded from the CPUs. TwinFin does not distinguish storage and data processing nodes, allowing for tighter integration of layers and, presumably, more optimized query execution.

Common to all these platforms is the combination of off-the-shelf components and a customized architecture that also includes specialized hardware. SwissBox is built in a similar fashion, but aims to explore in general the many cross-layer optimizations and alternative data processing techniques that become feasible in these new platforms.

1.2 Background: Use cases

SwissBox is motivated by real use cases provided by our industrial partners in the Enterprise Computing Center at ETH Zurich (`www.ecc.ethz.ch`). These emphasize the limitations of existing systems and suggest that a custom appliance could be a better approach to address them. Here, we focus on two such cases that underline the importance of cross layer optimizations.

The first is from Amadeus, the leading airline reservation provider. Amadeus acts as a cloud provider to the travel industry. Airlines, airports, travel agencies, and national authorities access reservation information through data services. Consequently, all queries have strict response time requirements (less than 2 seconds), and updates must also be propagated under tight time constraints. These guarantees must be maintained even under peak load (e.g., when a storm closes multiple airports in a region and a large wave of cancellations, re-bookings, and flight changes take place within a very short period). Furthermore, the query load does not consist solely of exact match queries: there is an increasing need for supporting complex queries under the same latency guarantees (for example, queries related to security).

As we have shown [16], the combination of tight latency guarantees and wide variety of loads cannot be handled with existing engines. The approach of materializing a view and using customized indexes for each data service quickly reaches a limit on scalability, not to mention the excessive maintenance cost associated with the full set of views and indexes necessary in a system of this size.

The challenge of the Amadeus use-case is combining OLTP and OLAP in a single system under very tight response time constraints. The requirements are as follows: deterministic and predictable behavior under a range of query complexity and update loads without the maintenance effort of complex tuning parameters, scaling to a large number of processing nodes, and enough reliability for continuous operation.

Existing commercial appliances address some, but not all, of these. All aim at reducing administration costs by removing tuning options: BWA automates schema organization and index creation, while TwinFin simply does without them. However, none of the commercial appliances we discuss provide, e.g., predictable response times or support for heavy update loads.

The second use case comes from the banking industry, and arises from the combination of increasing volumes of automated trading and the increasing complexity of regulations governing such trades. Algorithmic trading results in a high volume trade stream where the latency between placing the order and the order being executed is a key factor in the potential earnings. Existing systems can neither cope with the volume nor process the data fast enough to keep the latency to an acceptable minimum [13]. The need to inspect these real time streams for breaches of regulations, unauthorized trades, and violations of risk thresholds is a daunting but critical task.

The biggest challenge here is processing large volumes of data under tight latency constraints, with the processing involved becoming increasingly complex and sophisticated over time. The requirements we draw from the banking use case are: wire-speed data processing, low latency responses, dealing with data high volumes in real time, and scaling in both bandwidth and query complexity.

As above, commercial appliances at best address these requirements partially. Both TwinFin and Exadata emphasize the use of specialized networks to speed up operations, as the network quickly becomes the main bottleneck in such systems. Both appliances also move some processing to the storage layer (Exadata) or to network data path (TwinFin) to reduce the amount of data rate through the CPUs. However, none of these appliances support on-the-fly processing of data streams and they only provide hardware offload for simple operations.

1.3 Contributions

Motivated by these uses cases and the lack of a suitable solution, in this paper we outline SwissBox. SwissBox is a blueprint for data processing appliances that combines several innovative ideas into a single, tightly integrated platform. A key design principle of SwissBox is that extensive cross-layer optimizations (particularly across more than two layers) are a key high-level technique for meeting the requirements of modern application scenarios.

The importance of such techniques have been mentioned before. For instance, as part of the data stream processing carried out along complex and geographically distributed computational trees (an idea best captured by the Berkeley HiFi project [6]). In other areas of systems research, cross-layer techniques have also been the motivation for new designs of operating system [5] and proposals for changes to the Internet Architecture [11].

The appliance model allows cross-layer optimization to be applied to an entire stack within a single box. As such, the appliance model offers tremendous potential for gains from such optimizations. The corresponding challenge, however, is that essentially all layers of these systems need to be redesigned to both adapt them to the new hardware platforms and to open them up for a better interaction across layers, if their full potential is to be realized. The optimizations we propose in SwissBox represent an important step forward in the co-design of all systems layers.

In the short term, SwissBox is a practical way to address the challenges of predictability and scaleout presented by our use cases, in a way simply not possible with traditional database engines or cloud infrastructures. Longer term, SwissBox is a vehicle for exploring the wider possibilities opened up by the appliance concept. The model of custom configurations of COTS hardware within a closed appliance allows us to optimize across many layers in the software and hardware stack, by adopting novel designs of query processor, operating system, storage manager, hardware acceleration, and interconnect. We see SwissBox as a first step in this direction.

In the rest of the paper we describe our provisional design for the SwissBox architecture, and discuss lessons learned and research directions arising from our work so far.

2. SWISSBOX

SwissBox is a modular architecture for building scalable data processing appliances with predictable performance. The architecture consists of a flexible model of the underlying hardware, together with a functional decomposition of software into layers (though, as with network protocol stacks, this does not necessarily imply a layered implementation).

Many of the design decisions in SwissBox have been made both

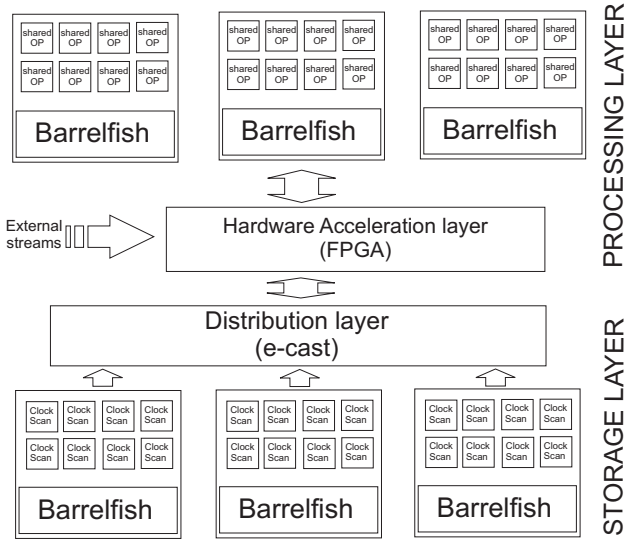


Figure 1: The architecture of SwissBox

to take advantage of current hardware and also to facilitate the adoption of new hardware developments as they become available. A SwissBox appliance consists of a cluster of commodity multicore computational nodes, each of which has multiple network interfaces attached to a scalable interconnect fabric. In turn, some of these interfaces will include programmable FPGAs on the datapath. SwissBox is agnostic as to the processor core types, and we expect a mix of general-purpose cores, GPU-like processors, and additional FPGAs to be used depending on the configuration. A promising candidate for the fabric is a 10 Gigabit OpenFlow [14] switch, which would provide plenty of opportunities for optimization across routing and forwarding layers inside the appliance, but Infiniband is a viable alternative. The fabric also has multiple external network ports for connecting the appliance to client machines.

The software architecture of SwissBox is shown in Figure 1. As in BWA and Exadata, SwissBox distinguishes between storage and data processing nodes. Storage nodes are multicore machines which implement the storage manager completely in main memory, removing the need for block-granularity operations and making it easier to adopt byte-addressable persistent storage technology such as Phase Change Memory as it becomes available. As in Exadata and TwinFin, storage nodes can themselves execute query operators close to the data. Reliability and availability is achieved by replicating the data across different nodes, while data partitioning provides scaling. The storage nodes also provide the basis for predictable performance by delivering tight guarantees on the time needed to access any data. Data processing nodes are also multicore machines configured as a cloud of dynamically provisioned resources. These nodes run complex query operators shared among concurrent queries and are deployed at the level of cores.

The layers in the software stack are described in detail in the sections which follow. At the lowest level, the Barrelfish research operating system [2] manages heterogeneous multicore machines and provides scheduling, job placement, interprocess communication, and device management to upper layer software. The *storage layer* is a distributed version of Crescando [16] (represented as a clock scan in each core in Figure 1), a main memory storage manager with precise latency guarantees that supports basic projection and selection as well as exact match and range queries.

The storage nodes are coordinated by a *distribution layer* using a novel agreement protocol call *e-cast* to enforce delivery and ordering guarantees while supporting dynamic reconfiguration. Between the distribution layer and the data processing nodes, a *hardware acceleration layer* uses FPGAs to pre-process data streams, building on our previous results in this area [17, 13]. Finally, the *data processing layer* implements high-level operators (join, sort, group-by, etc.) and scales using techniques such as result sharing across queries and orthogonal deployment of operators over cores.

3. OPERATING SYSTEM LAYER

An OS for SwissBox faces several key challenges.

One is traditional: there has always existed a tension between operating systems and database management systems (see, for example, Gray [9]) over resource management. To somewhat oversimplify, an OS is usually designed to multiplex the machine between processes based on some global policy, and present an abstract "virtual machine" interface to applications which hides the details of resource management. This is generally useful, but for databases it obstructs the application from making optimizations based on resource availability, and denies it the mechanisms to internally manage and multiplex the resources allocated to it.

Other challenges are specific to neither databases nor appliances, but are consequences of recent hardware trends. The advent of multicore computing, where the number of cores on a die now roughly follows Moore's law, has lead to scaling challenges in conventional OSes like Windows or Linux. Such systems are based on coherent shared memory, and suffer in performance as core count increases due to contention for locks, and (more significantly) the simple cost of moving cache lines between cores and between packages. Such scaling effects can be, and are, mitigated in traditional OS kernels, but at a huge cost in software engineering [2]. Worse, these expensive measures are typically rendered obsolete by further advances in hardware, requiring further re-engineering. for appliances.

Furthermore, traditional OS structures are simply unable to integrate heterogeneous processing cores (instruction set variants, GPUs, etc.) within a single management framework, nor will they function on future machines without coherent caches or shared memory, which are being discussed by chip vendors [10].

Commercial appliances use off-the-shelf OS kernels, which the vendors then specialize to their hardware platform. This takes advantage of the freedom given in the appliance model to optimize layers of software without undue concern for strict compatibility, and using commodity software dramatically reduces time-to-market. However, it does not fully meet the emerging challenges outlined above, most of which arise from the basic architecture of the OS. In addition, the approach does not exploit the opportunities to rethink the software stack across layers.

In contrast, the OS in each node of SwissBox is Barrelfish [2], a research operating system specifically designed to manage heterogeneous, multicore environments. Barrelfish is a "multikernel": an OS structured as a distributed system whose cores communicate exclusively via message-passing. This provides a sound foundation for scalability in the future by completely replacing the use of shared data structures with replication and partitioning techniques. By separating efficient messaging from core OS functionality, Barrelfish is less reliant on short-lived hardware-specific optimizations for multicore performance. As a distributed system, Barrelfish also easily handles heterogeneous processing elements and machines without cache coherence.

We also expect Barrelfish to be a better fit for data processing applications. On each core, the OS is structured as an Exokernel [5], minimizing resource allocation policy in the kernel and delegating management of resources to application as much as

possible. Barrelfish extends this philosophy to the multicore case through the use of distributed capabilities. The resulting OS retains responsibility for securely multiplexing resources among applications, but presents them to applications in an "unabstracted" form, together with efficient mechanisms to support the application managing these resources internally. To take two examples very relevant to databases, CPU cores are allocated to an application through upcalls, rather than transparently resuming a process, allowing fine-grained and efficient application-level scheduling of concurrent activities, and page faults are vectored back to applications, which maintain their own page tables, enabling flexible application management of physical memory.

Beyond this, the use of a flexible research OS within SwissBox opens up the exciting possibility of *OS-Database codesign*: architecting both the operating system and the data processing application at the same time. Such opportunities are rare, and the design of the appropriate interface between an OS and the data storage and processing layers in SwissBox is an exciting research direction. For instance, the SwissBox storage layer knows well the read/write patterns it generates, and we are exploring how to pass that information to the OS so that the OS can made intelligent decisions rather than getting on the way or having to "second guess" the application. Similarly, from those patterns the OS could derive copy on write optimizations to facilitate recovery and seamless integration of the main memory storage layer with a file system.

4. STORAGE LAYER

The storage layer in SwissBox is based on Crescando, a data management system designed for the Amadeus use case [16]. In SwissBox, we use a distributed version of Crescando as the storage layer because of its predictable performance and easy scalability. It also establishes the basis for exploring completely new database architectures that provide high consistency, exploit the availability of main memory, while still enabling the levels of elasticity and scalability needed in large applications.

In SwissBox, the interface to the storage manager is at the level of tuples rather than at the level of disk/memory blocks. Similarly, rather than simple get and puts over blocks, the storage is accessed via predicates over tuples. This greatly facilitates pushing part of the query processing down to the storage layer and makes the storage manager an integral part of the data processing system not just an abstraction over disk storage. This design choice is crucial to make SwissBox open to further hardware developments and to provide the same interface regarding of the hardware (or hardware layers) behind the storage manager.

Briefly described, Crescando works as a collection of data processing units at the core level. The storage manager works entirely in main memory, using clusters of machines for scalability and replication for fault tolerance. Each core continuously scans its main memory (hence the name *clock scan*), using an update cursor and a read cursor where queries and updates are attached in every cycle. Instead of indexing the data, Crescando dynamically indexes the queries to speed up checking each record against all outstanding queries. The latency guarantees of Crescando are determined by the time it takes to complete a scan. In current hardware, Crescando can scan 1.5 GBytes per core in less than a second, answering thousands of queries per cycle and core, even under a heavy update load. Within a scan, Crescando maintains the equivalent of snapshot isolation, with consistency across the whole system being maintained by *e-cast*, an agreement protocol specifically developed for Crescando (see below).

Crescando partitions the data into segments that are placed into the local memory of a single core. The segments can be replicated across cores and/or across machines, depending on the configuration chosen. For availability, Crescando maintains several copies of each segment across different machines so that if one fails, the segment is still available at a different node. Recovery in Crescando only requires to place the corresponding data in the memory of the recovered machine.

To understand the advantages of Crescando, it is important to understand how it differs from a conventional engine. For single queries over indexed attributes, Crescando is much slower than a conventional engine. However, as soon as a query requires a full table scan or there are updates, Crescando performs much better and with more stability than conventional engines. In other words, Crescando trades-off single, one at a time query performance for predictability (in Crescando all queries and updates are guarantee to complete within a scan period) and throughput (Crescando is optimized for running thousands of queries in parallel).

When compared to existing appliances, the storage layer of SwissBox offers several advantages.

As in Exadata, Crescando is an intelligent storage layer that can process selection and projection predicates, as well as exact match and range queries. This eliminates a lot of unnecessary traffic from the network and offers the first opportunity within SwissBox for pushing down operators close to the source. Unlike Exadata, in SwissBox we do not need an extra function shipping module or extra code at the storage layer. The storage layer is built directly as a query processor that can be easily tuned to match any latency/throughput requirements with two simple parameters (the size of a scan and the level of replication).

Like in SAP's BWA, the storage layer of SwissBox processes data in main memory, avoiding the overhead of data transfer from disk. In SwissBox, indexes are not used and the database administrator does not need to worry about identifying which ones are needed. This is an important point as one of the motivations for, e.g., SAP's BWA is to provide performance in spite of the fact that users are likely to chose the wrong indexes and data organization (and this is why BWA reorganizes the data). In the future, the main memory approach of SwissBox is ideally suited to quickly adopt Phase Change Memory, thereby combining the advantages of in memory processing with persistent storage.

In terms of cross layer optimizations, the design of Crescando fits very well with the premises behind Barrelfish. Crescando can use the interfaces provided by Barrelfish to make runtime decisions on placement and scheduling. The data storage layer can also be combined with modern networking techniques such as RDMA (Remote Direct Memory Access) [7]. Through RDMA, a node can place data directly in the memory of another node without involving the operating system and minimizing copying of the data. This is a feature that is commonly available in high end networks like Infiniband (the network used in Oracle Exadata) but that is now available through special network cards also for Ethernet networks. Using RDMA, recovery of nodes, data reorganization, and dynamic creation of copies can be greatly accelerated over traditional approaches. For instance, there is some initial work on rethinking the processing of data joins using RDMA [8].

As an alternative design option, we are also exploring using a similar one-scan-per-core approach but on a column store (as in SAP's BWA). Initial experiments indicate that column stores in main memory allow interesting and quite effective optimizations even if no disk access is involved. An intriguing idea that can be easily supported by the SwissBox storage layer is to store the data both row-wise and column-wise, routing queries to the most appropriate representation.

5. DISTRIBUTION LAYER

The distribution layer of SwissBox is one of its distinct features that differentiate it from existing appliances and that confers SwissBox very interesting properties. Such a layer is commonly found in cloud solutions, particularly in distributed key value stores, e.g., [1], but we are not aware of any database appliance using anything similar.

In SwissBox, this layer is implemented on the basis of a new agreement protocol, e-cast, that distributes reads and writes across the storage layer nodes. E-cast combines results from virtual synchrony [3] and state machine replication [12, 15] to provide a highly optimized protocol that maintains consistency across replicated, partitioned data. Aside of the performance and configuration advantages it offers, e-cast supports dynamic reorganization for both transparent fail-over and elasticity. Through these features, SwissBox is in a position to easily scale up and down its storage layer, something that is not possible as far as we are aware with existing appliances.

The distribution layer offers many opportunities for research and exploring new architectural designs. For instance, E-cast works across nodes of the storage layer. Barrelfish does something similar but across the cores of a multicore machine. As part of ongoing work, we are analyzing the characteristics of multicore machines as distributed systems and taking advantage of their special properties (e.g., messaging implemented as shared memory, synchronized clocks) to develop a suite of agreement protocols specially tailored to multicore machines. An interesting design option would be to interface e-cast and Barrelfish so that the two of them act as seamless extensions of each other. This is particularly interesting in SwissBox because both the storage management and data processing layers are organized in terms of independent units of execution allocated to one core. The resulting protocol would act as a hierarchical network that would allow SwissBox to treat a pool of multicore machines not as a series of independent nodes but as a pool of cores with the distance between them (the network overhead) as the parameter to use for query optimization.

The elasticity provided by e-cast will also play a crucial role in the automatic configuration of SwissBox. As pointed out above, a key parameter in tuning the response time of the storage manager is the size of a clock scan. The smaller the scan, the faster the storage layer but the more nodes are needed. With e-cast we have a way to easily reorganize the storage layer: consolidating the data in few nodes if the response time constraints allow it or expanding the storage layer across many nodes if the size of the scan needs to be reduced or the level of replication needs to be increased to meet the given performance requirements. E-cast provides here the logic to maintain consistency in the face of dynamic reconfigurations while techniques like RDMA provide the mechanisms for quickly migrating, copying, or reorganizing the independent work units allocated to cores.

6. HARDWARE ACCELERATION LAYER

Both the use case from the financial industry discussed above and the emphasis of commercial appliances in fast networks point to one of the major bottlenecks encountered today by cluster based solutions: the network. SwissBox is no exception in this regard. The problem is twofold. On the one hand, concurrent accesses compete for bandwidth and the network latency is higher than on a local disk. On the other hand, using a network card typically involves the operating system, additional data copies, and a reduced capacity to process data at the speed it arrives from the network.

In SwissBox we use an additional layer of FPGAs for processing the data streams as they travel from the storage layer to the data processing layer. The hardware acceleration layer offers a second tier to which operators and parts of queries can be pushed down. This layer is similar in functionality to that found in TwinFin and can be used for a variety of purposes: aggregation, decompression, dictionary translation and expansion, complex event detection, or filtering of tuples. The key advantage of this layer is that we have shown it can process data at wire speed, something that conventional network interfaces cannot do unless the number of packets from the network is drastically reduced [17].

An interesting aspect of the hardware acceleration layer is that the implementation of data processing algorithms in hardware requires very different techniques than those commonly used in CPU based systems. For instance, in [17] we have shown that the functionality of a hash table can be implemented in hardware with a pipelined circuit that can process multiple elements in parallel, reaching a much higher throughput than in a CPU based system. Such fundamental differences in the underlying algorithms raise questions about the ability to compile queries where operators will be placed in different tiers and one of those tiers involves hardware based operators. Both Exadata and TwinFin claim that they place operators on different layers (the intelligent storage of Exadata, or the FPGAs of TwinFin) although the query compilation process and the cost optimization functions that guide the query compilation are not obvious. From the available information, these systems place only simple operators such as selection, projection, and decompression in the lower layers. A question that we aim at answering as part of the evolution of SwissBox is whether more complex operators can also be pushed down to the lower layers and what are the cost models to use as part of the query optimization process [13]. To do this, an ongoing research effort around SwissBox is how to characterize data processing operators so that we can provide a uniform interface to those operators regardless of where they are located in the data processing hierarchy within an appliance.

7. DATA PROCESSING LAYER

The layers described so far can only execute relative simple queries. To support the full of SQL or even generic data processing operators written in other languages, SwissBox incorporates a data processing layer where operators for more complex queries are executed. These operators are allocated to cores as independent work units in nodes running Barrelfish (in a similar configuration as the storage nodes). The scalability and elasticity in the design comes from the fact that the operators can be deployed in arbitrary cores/machines (similar to TwinFin and BWA), with the placement controlled by the query plan optimization. Unlike in existing systems, these operators do not work one query at a time. They have been designed to be shared by many concurrent queries (in the thousands) and make heavy use of intermediate result sharing. The way this is achieved is by using the Crescando storage nodes –which also process queries concurrently– to label a stream of results with the id's of the queries for which the record is intended. These result sets are streamed to the corresponding data processing node (after filtering through the FPGA layer), which then performs the operation concurrently for all queries, separating the results only afterwards. This design increases the possible throughput considerably and also helps to reduce the data traffic across the system since records that are in the result set or intermediate result set of many queries are sent only once rather than once for each outstanding query.

Note that the data processing operators can be placed on data processing nodes or they can also be placed on the core of storage nodes. This gives SwissBox another degree of freedom for deploy-

ing highly optimized query plans. For instance, operators such as sort, group by, and aggregation can be easily placed on cores next to Crescando cores. The result is that there is no traffic over the network as, in many cases, the entire query can be pushed down to the storage layer. This design mirrors somewhat that of TwinFin in that it blurs the separation between data processing and storage nodes. Unlike TwinFin, however, we can eliminate the traffic over the network entirely for some queries and we can choose to place the operators in the data processing nodes instead for scalability and elasticity purposes.

The advantage of the data processing layer approach is that we can maintain the predictability of the system by ensuring the execution of a part of a query that runs on these nodes has a well defined upper bound. The overall cost of a query would then be the time to get the data from the storage layer (which is accurately defined thanks to the Crescando approach), the transmission overhead of the network if any, and the processing overhead of the high level SQL operators in the data processing nodes. We believe that the unique architecture of SwissBox allows to place tight bounds on these overheads and to capture these bounds with very few parameters. This opens up the possibility of tools for automatic configuration of all layers of SwissBox given a set of response time requirements. Note that such tools will make heavy use of the open interface of Barrelfish, the well defined tuning parameters of Crescando, and the elasticity of the data processing layer, with the hardware acceleration and the possibility of pushing down operators close to the storage layer providing additional leverage to address the problem.

8. CONCLUSIONS

SwissBox is intended both as a data appliance to be deployed in real settings and as a vehicle for research to enable the complete redesign of the software stack.

As an appliance, SwissBox has a number of unique features that make it very suitable to the use cases described earlier as well as in a wide range of other environments. In SwissBox, there is no locking or read-write contention at the storage layer, allowing us to support very high update rates without impacting the read rates. Moreover, all layers of the system but specially the storage manager –which is traditionally the main problem in this regard– provide predictable performance for both reads and writes. This allows us to build systems that by design can meet tight response time constraints.

When compared to key-value stores like Cassandra [1], SwissBox provides full data consistency, elasticity, and the ability to perform complex SQL queries, thereby establishing a completely new point in the design space. When compared to existing appliances, SwissBox's multi-query optimization strategies at all levels and the organization of work into independent units mapped to cores provide a degree of flexibility and scalability that cannot be achieved with disk based systems. In SwissBox we can benefit from many of the same hardware optimizations used in existing appliances (e.g., SSD storage, FPGA processing) but we can also exploit many other cross-layer optimizations that are not possible in today's appliances thanks to the use of Barrelfish as the underlying operating system.

As a research platform, SwissBox gives us the opportunity to completely redesign the data processing stack from the ground up, exploring at the same time how to best take advantage of new developments in hardware. This is a rather urgent matter given the pace at which key elements of the data processing infrastructure are evolving. That SwissBox is an appliance makes it possible to apply cross-layer optimizations within a single box, thereby opening up the possibility of taking these cross-layers optimizations much further than it has been possible in distributed platforms. At the same time, it will allow us to rethink the interfaces and role of the different layers of a data processing system in terms of meeting new requirements like predictable performance or elasticity. Our intention is to make SWissBox open source -even the hardware architecture- to provide an open platform for experimentation and education where new data processing techniques can be tested and compared free from the limitations and legacy constraints of existing database engines.

9. REFERENCES

[1] *http://cassandra.apache.org/.*

[2] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, pages 29–44, 2009.

[3] K. Birman. A History of the Virtual Synchrony Model. Technical report, Cornell University, 2009. http://www.cs.cornell.edu/ken/History.pdf.

[4] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. SOSP*, pages 251–266, December 1995.

[6] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. W. 0002, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, pages 290–304, 2005.

[7] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *ICDCS*, pages 553–560, 2009.

[8] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *ICDCS*, pages 283–292, 2010.

[9] J. Gray. Notes on database operating systems. In Bayer et. al., editor, *Operating Systems, an Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[10] J. Howard and et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference*, pages 108–109, Feb. 2010.

[11] R. R. Kompella, A. Greenberg, J. Rexford, A. C. Snoeren, and J. Yates. Cross-layer visibility as a service. In *In Proc. IV HotNets Workshop*, 2005.

[12] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 6(2), 1984.

[13] R. Müller, J. Teubner, and G. Alonso. Streams on Wires - A Query Compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.

[14] OpenFlow Consortium. Openflow. `www.openflow.org`, September 2010.

[15] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(299), 1990.

[16] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, 2009.

[17] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB*, 3(1), 2010.

IQ: The Case for Iterative Querying for Knowledge

Yosi Mass[1,3] Maya Ramanath[2] Yehoshua Sagiv[3] Gerhard Weikum[2]

[1]IBM Haifa Research Lab
Haifa, Israel
yosimass@il.ibm.com

[2]Max-Planck Institute for Informatics
Saarbrücken, Germany
{ramanath,weikum}@mpi-inf.mpg.de

[3]The Hebrew University
Jerusalem, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Large knowledge bases, the Linked Data cloud, and Web 2.0 communities open up new opportunities for deep question answering to support the advanced information needs of knowledge workers like students, journalists, or business analysts. This calls for going beyond keyword search, towards more expressive ways of entity-relationship-oriented querying with graph constraints or even full-fledged languages like SPARQL (over graph-structured, schema-less data). However, a neglected aspect of this active research direction is the need to support also query refinements, relaxations, and interactive exploration, as single-shot queries are often insufficient for the users' tasks. This paper addresses this issue by discussing the paradigm of Iterative Querying, IQ for short. We present two instantiations for IQ, one based on keyword search over labeled graphs combined with structural constraints, and another one based on extensions of the SPARQL language. We discuss the suitability of these approaches for knowledge-centric search tasks, and we identify open research problems that deserve greater attention.

1. INTRODUCTION

Advanced users such as journalists or analysts have information needs which are often expressed (even in natural language) as a mix of vague, precise, and implicit requirements. For example, consider the following queries: i) "I want to know something about classical music composers who have composed music for western movies." ii) "Could the H1N1 vaccine interfere with blood-pressure medications such as Metolazone?" iii) "How are Israel and Italy related to each other, for example, by some international organizations?"

Although Web search engines now have limited and specialized support for natural-language questions and are moving towards more expressive entity-oriented search (e.g., by understanding product names or locations), the above questions cannot be answered easily. Recently emerging knowledge engines [5, 4] or knowledge-base search services such as WolframAlpha[1], Google Squared[2], or *sig.ma* cannot cope with such complex queries either. Some of them seem to perform entity-oriented information extraction on-the-fly, while others harness large knowledge bases such as DBPedia[3], Freebase[4], True Knowledge[5], or the CIA WorldFactbook[6]. These contain billions of RDF triples about entities and relationships, but cannot retrieve the necessary facts for answering the above queries or lack inferencing capabilities for composing proper answers. For example, trueknowledge provides a browser plug-in that supplements keyword-search results from Google or Bing with fact-retrieval answers from their knowledge base. This is good enough for returning the correct birth place of Barack Obama, but still far from handling our examples.

So neither Web search engines nor knowledge-base engines can directly answer such advanced questions. However, there is often a solution if the user is willing to engage in an entire workflow of query refinement, query relaxation, exploration of intermediate results, combining different results, etc. This is tedious but often works. The point is that, instead of running a single-shot query, we need a process of *iterative querying*, *IQ* for short. In fact, this is what search-engine users often end up doing, but there is not much support by the engine. Moreover, while IR researchers are advocating interactive retrieval for many years [13] (without compelling impact), in the structured-data world of knowledge bases and inference engines, the expectation by DB folks is that everything can be expressed in a single query of some super-powerful language (be it SQL, XQuery, SPARQL, or whatever).

An IQ task with a Web search engine would involve the following steps:
i) *exploration*: retrieving initial results by keyword search,
ii) *filtering*: refining the results with additional constraints,
iii) *aggregation*: combining results into a concise answer.

These steps may themselves have to be iterated. For example, for the composer question, one could proceed as follows: i) find a list of classical music composers; again i) find a list of composers for western movies; ii) pick out composers of western movies who also compose classical music; once more i) find biographies for each of the interesting composers; ii) ensure that the biographies match the list of com-

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

[1] *wolframalpha.com*
[2] *google.com/squared/*
[3] *dbpedia.org*
[4] *freebase.com*
[5] *trueknowledge.com*
[6] *www.cia.gov/library/publications/the-world-factbook/*

posers (as keyword search can easily return wrong results); iii) aggregate multiple pages about the same composer into a compact summary.

For the world of structured knowledge bases, this kind of IQ process seems totally neglected so far and widely open for research. Knowledge portals such as dbpedia.org or freebase.com offer APIs that support only single-shot querying, by means of SPARQL calls; their UIs, on the other hand, are merely Web-pages with fancy rendering but tedious navigation for the user. The irony is that despite semantically structured data, there is poor support for advanced questions about factual knowledge.

In this paper, we advocate the IQ paradigm for search against *semantic knowledge bases* or the linked-data cloud (*linkeddata.org*) [3] of structured data on the Web. We discuss the requirements for the exploration, filtering and aggregation steps, describe the development of effective tools in two case studies and identify challenges for future research.

2. DATA MODEL

The diversity of data available in different kinds of knowledge bases, linked data on the Web, as well as text in the form of contextual information, requires a very general and flexible data model. The natural choice is to use a directed graph where each *node* is allowed to have: a *name* (identifier or short string describing, for example, an entity name or a paper title), a *type* (the class to which the node belongs), and a *context* (additional text that is associated with the node). Similarly, each *edge* is allowed to have: a *name* (identifier), a *type* (a label denoting, for example, a relationship type), and a *context* (textual information, for example, denoting the context in which the fact denoted by the edge and its two end points was extracted from).

This unified model captures all the different approaches to graph search. It can represent, for example, XML (nodes labeled, edges unlabeled), RDF triples (nodes and edges labeled), relational databases (records as nodes, foreign-key relationships as edges), etc. Note that in general, there is no schema for these knowledge bases, apart from a generic triple representing an edge in the data graph (referred to as subject, predicate and object in RDF).

3. THE IQ PARADIGM

As in the case of current day Web search, we envision iterative querying as being composed of a combination of three steps: exploration, filtering and aggregation. However, unlike Web search where there is no structure at all, and unlike relational databases where the schema is fixed, in our data model there are entities (nodes) and relationships (edges), but there is no fixed schema. Therefore, the exploration, filtering and aggregation steps of the IQ paradigm should allow the user to work in two dimensions. First, the user should be able to select desired structural patterns (i.e., schemas) of answers and secondly, she should be able to select the relevant instances of those patterns.

Figure 1 illustrates our framework. Users query data in the underlying knowledge-bases through a UI. A system built on the IQ framework and supporting the exploration, filtering and aggregation steps interacts with the API and returns results back to the user. The goal of an IQ system should be to automate as many tasks as possible, and to involve the user only at certain critical steps when her

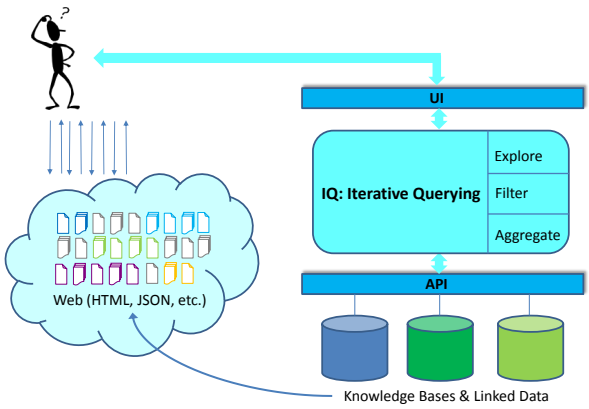


Figure 1: The IQ Paradigm

feedback is essential.

Clearly, while the UI has to be simple and intuitive, the API should ideally be expressive enough to handle complex tasks, including aggregation. The IQ system provides a bridge between the UI and API, and in certain cases may have to extend the API to support the needed functionality.

In the rest of this section, we describe the key ideas of each step, and defer technical details of how they could be achieved to the case studies in Sections 4 and 5.

Exploration. The first step is to help the user express her information need in a precise manner. A natural way to start the exploratory step is for the user to enter some relevant keywords, or phrases and for the system to return possible connections between them. For example, for the query on classical composers, a starting point is for the user to enter "classical music" "composer" and "westerns" or for the query on the relationship between Israel and Italy, to enter "Israel" and "Italy". It is now up to the system to search for possible connections between these keyphrases.

Given a set of diverse connections from the underlying data graph, the user would prioritize certain structural constraints while disallowing certain other kinds of connections. The system repeats the query with these additional (positive and negative) constraints. At the end of this iterative process, a precise query of the form "classical composer X composed for western Y" is formulated, where X and Y have to be substituted with a person and a movie, respectively. This query could then be mapped onto the API for a structured query language.

However, this may not be the end of the exploration step. Perhaps there are other closely related queries such as "classical *musician* X composed for western Y" or "classical *music conductor* X *directed music* for western Y", which can additionally be suggested to the user. Once the user converges on a subset of precise queries as representative of her information need, the system can now move on to the filtering step.

Filter and Refine. While the exploratory step helps users narrow in on possible interesting queries, the filtering step allows users to specify exactly which items in a result set are interesting. The system runs the queries formulated in the

previous step directly on the knowledge-base and returns a set of qualifying results. The user now refines her query in order to filter out the results of interest to her. The filtering could be as straightforward as selecting certain interesting results, or there may now be additional refinements (in the form of constraints) to restrict the size of the result set. For example, movies with a certain plot-line (say, involving "soldiers"), or musicians born in a particular continent, etc. Additionally, after seeing this set of results, the user may also disallow certain queries formulated in the previous step or give additional weight to certain other queries. Iteratively processing these refined queries may finally lead to a satisfactory result list. Alternatively, the user could repeat the exploration step if the results are still unsatisfactory to her.

Aggregation. Our data model (Section 2) is general enough to accommodate a wide variety of data including a combination of structured triples and unstructured text. Aggregating results can thus take on different meanings, ranging from a simple ranking of results (famous composers first), grouping of results (composers grouped by their nationalities), diversifying the top results, etc. to type-specific aggregations such as summarization of text (for example, biography summarization for composers).

Aggregating results from a single knowledge-base is complex enough by itself, but the situation becomes more complicated if there are multiple knowledge-bases which need to be queried. A likely scenario is that the local knowledge-base has insufficient information to answer the query, but has multiple pointers to other knowledge-bases (in the linked-data spirit) which are likely to contain the missing information. The system now has to decide whether and how the query should be split (perhaps there is overlapping information in the knowledge-bases which can be leveraged) and subsequently, how to merge the (partial) results. Both tasks are non-trivial because on-the-fly entity disambiguation [9] may be required due to different vocabularies.

With this overview of the IQ paradigm, we now illustrate two case studies of how a system built on these principles would work. In the first case study, the user starts the exploration step with SPARQL queries, and refines queries by adding keywords to the structured queries. In contrast, in the second case study, the user initially does not have any knowledge about the structure of the data graph. Hence, she starts the exploration of the knowledge-base with keywords, while the refinement step involves selecting and unselecting suitable structures showing the interconnections between the keywords. The selected interconnections can then be converted to SPARQL to further the search. Finally, the aggregation step for each case study highlights a different kind of aggregation—ranking and grouping of results in the first study, and merging results from multiple knowledge-bases in the second. In principle, however, both types of aggregation are applicable to either one of the two case studies.

4. CASE STUDY 1: IQ WITH EXTENDED SPARQL

The W3C-endorsed query language SPARQL is a natural starting point for searching knowledge bases or the world of linked-data Web sources. SPARQL is designed for structured RDF data, but does not need a prescriptive schema for its data, and can cope with high heterogeneity. The basic building block in a SPARQL query is a triple pattern: essentially an SPO triple with one or several of the S, P, and O components replaced by variables. Multiple triple patterns are combined in a conjunctive manner, thus supporting select-project-join queries. The key point, compared to traditional database querying, is that even properties—the counterparts of relation or attribute names—can be variables. For example, when searching for composers of film music, we may not know how exactly the relationship between composer and movie is named, e.g., composed, isComposerOf, wroteMusicFor, contributedToSoundtrack, etc. Or the data may be so heterogeneous that no single property name is suitable for high recall. With SPARQL, we could express a sub-query for the composers question as follows:

```
SELECT ?c, ?m
WHERE {
  ?c hasType composer . ?m hasType movie .
  ?m hasGenre Western . ?c ?prop ?m . }
```

where variables start with a question mark and the appearance of the same variable in different triple patterns denotes a join condition.

Exploration. Despite the schema-agnostic option for query formulation, users (or programmers on behalf of users) still need some awareness of the underlying RDF structures. If composers are first related to their compositions, say by a property composedPieceOfMusic, and the compositions are in turn related to movies by a property appearedInSoundtrackOf, even the wildcard pattern ?c ?prop ?m would not return any matches as all variable bindings need to come from a single RDF triple. Fortunately, it is not too difficult to extend SPARQL, to support these cases while preserving the general flavor of SPARQL. Following the proposal by [2], we could introduce variables that can be bound to entire paths in the RDF triples graph. Moreover, as we do not simply want connectivity but have semantic requirements, we would combine this with filter conditions on the property names in the qualifying paths. For our example, this could be phrased as follows:

```
SELECT ?c, ?m
WHERE {
  ?c hasType composer . ?m hasType movie .
  ?m hasGenre Western . ?c ??prop ?m .
  Filter regex(??prop, {"compose"}) .
  Filter pathlength(??prop, 3) . }
```

where ??prop is a path variable (note the two question marks), regex is a regular expression (simple substring matching in our case) on the path of property names bound to ??prop, and the last condition sets an upper bound on the path length. The Filter construct is standard SPARQL, to include conditions beyond exact-match on URIs or literals. Here we deliberately use it for extensibility.

Further, the system could support *query relaxation*, and suggest close-and-related queries. For example, for the swine-flu vaccine question, a user formulation like

```
SELECT ?c, ?d
WHERE {
  ?c hasType H1N1vaccine .
  ?c interferesWith ?d.
  ?d hasType BPMedication . }
```

could be automatically relaxed into:

Figure 2: Western movies: i) filtered by "soldier", ii) unfiltered

```
SELECT ?c, ?d
WHERE {
  ?c hasType H1N1vaccine .
  ?c (interferesWith|notRecommendedWith|createsRiskWith) ?d .
  ?d hasType BPMedication . }
```

Filter and Refine. The exploratory step works reasonably well when everything the user wants can be expressed in triple patterns, but is somewhat inflexible. It may not be possible to express certain kinds of constraints using just triple patterns, or the user may not know how to (for example, Westerns with a particular plot line involving soldiers). On the other hand, the knowledge base itself does not have every conceivable property. For example, there may be a lot of facts about compositions, but no predicates with classical music.

The key to overcoming this obstacle is to extend the knowledge base with textual contexts from the Web. For every RDF triple in the database, we can reach out to the Web and gather text snippets where the triple occurs.

Now we associate words and phrases from these textual "witnesses" with each triple, and make them queryable as if they were a fourth dimension added to the three SPO dimensions. Inspired by XQuery Full-Text, we have developed such a SPARQL extension with the following specific syntax [6, 7]:

```
SELECT ?c, ?m
WHERE {
  ... ?c composed ?m{"classical music"} . }
```

where "classical music" is a phrase to be matched in one or more of the witnesses for the triples that qualify for the triple pattern. Similarly, filtering Westerns with a particular plot line involves adding the appropriate keywords. An example from our system [7] is shown in Figure 2—the order in which results are shown differs on whether a filter condition ("soldier") has been added to the triple pattern or not.

However, it now seems that the user must be familiar with the specific terminology "classical music" in the witnesses. But, the query relaxation for SPO patterns introduced previously, can naturally be extended to keyphrases as well if the user wishes to explore this. First, we could offer a dialog to the user with suggestions on related words and phrases such as "operas", "symphonies", "cantatas", "string quartets", etc. In IR this is known as query expansion; it can be done with explicit user involvement, or transparently to the user. The semantics for a qualifying triple is that its witnesses should contain at least one of the text terms, not necessarily all. We can plug in a suitable ranking model based on IR principles.

Aggregate. The results of the previous steps may overwhelm the user with too many answers (especially if query relaxation is used). Therefore, it is crucial to aggregate the results into an easily digestible form. Grouping and ranking are obvious ideas, and can even be utilized together.

Different groups (clusters) have different statistical evidence for their validity and informativeness. In the example question about swine-flu vaccines, there are definitely many conflicting sources, and a good answer needs to be backed by statistical mass and/or authoritative sources. This issue is important for aggregation into groups, but also shows up for ranking individual answers. A good answer needs to reflect salient entities and properties rather than exotic facts about long-tail entities. Estimating salience is non-trivial, especially if the corpus does not have redundancy—that is, confidence in a fact can no longer be estimated using frequency measures on the corpus.

Moreover, if the system has to handle "close-but-related" queries as well as textual conditions (both of which users can specify in the filter-and-refine step), developing an efficient ranking mechanism which integrates all these features becomes challenging. Recent efforts in this direction include [6, 7].

5. CASE STUDY 2: IQ WITH GRAPH-BASED KEYWORD SEARCH

In the previous case study, the user needs to have some knowledge about the structure of the data graph. For instance, in the composer example above, the user needs to know that there are triples matching the pattern ?c ?prop ?m. Even with the extension ?c ??prop ?m that supports path variables, it is still assumed that the path is from a composer (which is bound to ?c) to a movie (which is bound to ?m). It is possible, however, that the data graph is heterogeneous and in some cases, the edges are reversed, namely, there are triples matching ?m ?prop ?c. In such cases, the path variable will not assist in finding existing answers (unless more patterns are added). In summary, it is rather hard to start a search with the approach of the previous case study, when the user lacks any knowledge about the structure of the data graph.

In this case study, we assume that the user has either no knowledge or a very limited one about the underlying data graph. Thus, the *query* is just a set of keywords. The API may include, in addition to those keywords, some control characters for advanced search. From the user point of view, the search is carried out in two dimensions, namely, the types (i.e., schemas) of answers and their particular instantiations. For example, for the keywords "Italy" and "Israel," there are several types of answers. One is that the two countries are members in some organization. Another type is that the two countries are located on the shores of the same sea.

The data model is a directed graph, as described in Section 2. An *answer* is a *non-redundant* subtree t of the graph, such that t *contains* all the keywords of the given query. Containment means that each keyword may appear anywhere in the tree, that is, in the name, type or context of a node (or an edge). Non-redundancy means that an answer does not have a proper subtree that also contains all the

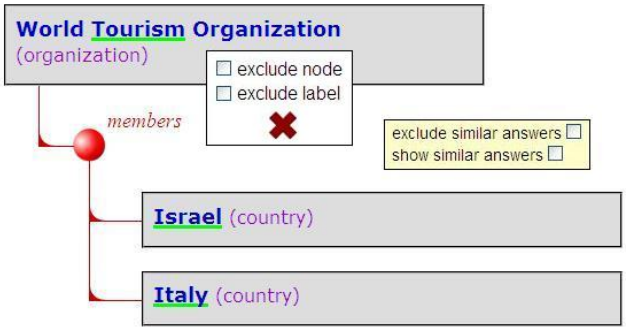
keywords of the query. Note that non-redundancy does *not* imply minimality, and a query could have a large number of answers. The *schema* of an answer is the tree obtained by ignoring the names and contexts, that is, each node (and edge) has only a type.

In this section, we consider the system demonstrated in [1], which is based on [8]. There are other systems for keywords search over data graph [11], but only [1] facilitates search in both dimensions, namely, answers and their schemas.

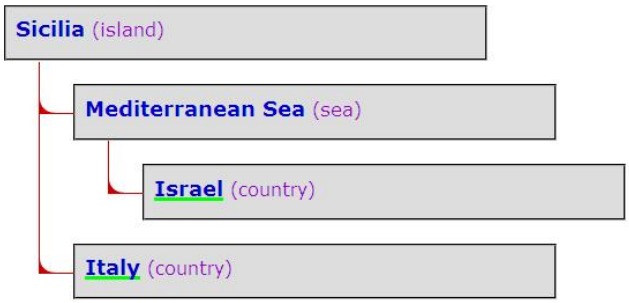
Exploration. A major problem with typical keyword search over data graphs is that the user may be inundated with too many answers that have the same "flavor" (i.e., schema). For example, both Italy and Israel are members in many international organizations. Hence, for the query "Italy Israel," there are going to be many answers—one for each organization. The user might have to browse through many pages until she gets something different. In [1], there are search options that enable the user to zoom in on the "flavors" of her choice. In particular, the user can choose an answer and specify that either she does not want to see more similar answers or all she needs are additional similar answers ("similar" means "with the same schema"). She can also specify that subsequent answers should not include particular names or types (i.e., labels). Figure 3(a) shows an answer for the query "Italy Israel." Note that names are capitalized, whereas labels (identifying types) are not; in addition, a label is shown inside parentheses when there is also a name. The schema of the answer, in Figure 3(a), connects two countries through membership in some organization. (In the data model of [1] only nodes have names, types and contexts.) By checking the option "show similar answers," the user can browse through all organizations in which both Italy and Israel are members. Alternatively, the user can exclude the labels "organization" and "members" so that subsequent answers will not include them. Consequently, the user will immediately get an answer that does not show any information about members of organizations. One such answer is shown in Figure 3(b), where the two countries are linked due to the fact that Israel and an island of Itlay are located on the shores of the Mediterrannean Sea. There is a subtle difference between choosing the option "exclude similar answers" versus excluding the labels "organization" and "members." The former, but not the latter, allows subsequent answers to be about organizations and members as long as their schemas are not identical to the one already seen.

In [1], there is also a mechanism for diversifying answers. The main idea is to apply adaptive ranking that takes into account similarity to answers that have already been shown to the user. Two answers are similar if they have the same schema, or if they have two subtrees that are identical or have the same schema. The ranking function is augmented with a parameter that takes the degree of similarity into account, and as a result, the next page of answers is likely to show results that are substantially different from previous ones. This mechanism does not require user intervention (other than turning it on). The user, however, has the option of tweaking the parameters that measure similarity.

A query may contain keywords that do not appear in the data graph. For example, the user is interested in "Italy" and "Israel" in the context of "music," but the data graph only contains the first two keywords. Similarly to Section 4,



(a) Members in the same organization



(b) Located on the shores of the same sea

Figure 3: Two answers with diverse schemas for the query "Italy Israel"

we can extend the data graph with textual context from the Web. But now there is a need for a suitable ranking function that also takes the context into account. One approach is to first explore the data graph without the context, while ignoring the keywords of the query that appear only in the context. Once some specific schemas are selected, the ranking takes all the keywords of the query into account and uses textual "witnesses" (as described in Section 4). However, this approach may not be as effective and quick as an exploration that uses all the keywords from the outset. An alternative is to extend the ranking technique of [8] by applying IR methods to the textual context.

Filter and Refine. The exploratory options described above (e.g., "show similar answers") are effective in practice, but are quite rudimentary. A more expressive way is to use trees (of answers produced thus far) in order to create SPARQL queries, similar to the approach of Section 4. There is a natural corresponds between a tree and a conjunction of triple patterns. There is, however, some latitude when creating triple patterns from a tree. We can use the names of the nodes and edges of the given tree, thereby creating a conjunction that matches just that particular tree. Alternatively, for some nodes and edges, we can take just the type or use a variable, and consequently, the generated SPARQL query would match many answers. We can also use the contexts of the nodes and edges in order to create additional selection criteria, as described in Section 4. The user need not be aware of this translation into a SPARQL API, because the system would do it automatically based on some interaction with the user through a high-level UI. Note that the SPARQL queries of Section 4 correspond, in

general, to graphs and not just trees. Thus, the user can actually construct a SPARQL query corresponding to a graph, based on the insight she has obtained from several answers with diverse schemas.

Aggregate. We focus here on how to combine results that come from different data sources. As an example, consider the query "find movies that were shot in countries that border Italy." We can decompose it into two sub-queries. (1) "?X = find countries that border Italy," and (2) "find movies that were shot in ?X." The first sub-query could be executed on the CIA World Factbook, and the second—on the IMDB (i.e., Internet Movie Database).

In general, the challenge is how to decompose the original query, and how to join the results of the sub-queries. One approach is to add support in the UI for decomposing the query into several subsets of keywords, according to the available data graphs, and join the results of the sub-queries as follows. Initially, the user executes the first sub-query, which is "countries that border Italy" in our example. Once she zooms in on the relevant answers, she selects specific nodes that are used for creating instantiations of the next query. In the above example, each country returned by the first sub-query is used to instantiate ?X in the second sub-query "movies shot in ?X."

6. CHALLENGES AND OUTLOOK

The IQ paradigm for search is a natural consequence of the complexity of the information needs of users as well as the underlying data. There has been a considerable amount of research on the many different aspects that a system built on this paradigm should support. Indeed, examples of these were illustrated in our case studies. However, there are still several hard problems which need to be solved. We list a few of these below.

Exploration. In order to make user interactions as easy as possible, the system has to understand natural-language questions—a hard problem on which some progress has been made in the last few years (see, for example, [10, 12]). The questions need to be mapped to a query language in order to make the processing more precise and efficient, but there will be parts of the question which may not map to any structure. We believe that data models such as the text-augmented RDF where both the structured, as well as unstructured text are queriable in a unified manner can help in more robust translations. One way to leverage this is, for example, to map the question into structured triple patterns when possible, and leave the rest as keywords attached to certain triple patterns. The triple patterns themselves may in turn be automatically derived by doing a keyword search on graphs and extracting the most interesting patterns from it.

Even in the case of expert users who may enter formal queries directly, there is still the issue of query correctness and whether the query will return a non-empty result set. Interactions in the form of close-but-correct query suggestions may be needed to guide the user.

Deriving these queries over a single knowledge-base is hard enough, and even more challenging is the case when there are several (possibly overlapping) sources. Choosing the appropriate set of sources itself becomes difficult, unless the user is in the loop.

Filter and Refine. Learning the history of the user alleviates user frustration, by providing, for example, a personalized ranking of results, from which the user can quickly select interesting ones. Moreover, answers can be tuned based on whether the user asking a query is an expert or a child.

On-the-fly personalization, where the system learns as the user goes through the search process is also a challenge. For example, the query on composers of western music could return a long list. But as the user chooses one or two composers who also compose classical music, the system could immediately change the ranking and return classical composers higher up.

Aggregate. The notion of whether two results are identical is important in some of the aggregations that we mentioned. For example, we may need to perform joins on results returned by two different knowledge sources, or group near-duplicate results into clusters.

Even in cases of joins involving just entities, there has to be an implicit entity disambiguation step. Are the two attributes the same entity despite having different names, or are they different entities despite having the same name?

In the universe of knowledge rather than Web pages, the notion of duplicates is more involved. For example, we may find the same composer related to operas in one answer and to symphonies in another answer. Since both operas and symphonies are considered classical music, are these two results duplicates? Apart these semantic issues, there is also no support for grouping in SPARQL, and the notion of grouping in text-augmented RDF data is totally unexplored. We doubt that generic clustering methods are suitable here.

User Interface. UI issues play a big role in identifying the basic functionality of the system. For example, consider the connection between doing aggregation of results and the display of results. Suppose a user asks for countries bordering Italy. Rather than seeing one answer for each bordering country, she would like to see all the bordering countries in one answer. However, if the user also wants a typical food recipe for each country, then aggregating all the countries and their recipes in one answer might create a display which is too cluttered. In general, there is a tradeoff between many similar answers and one aggregated answer. The former has a simple and clear display of each answer, but the list of those answers could be long. An aggregated answer might be too cumbersome to display neatly. So, it is not clear how aggregation can be carried out automatically at the level that is most suitable to the user.

Not all browsing and searching happens while a user sits at her machine, nor is every user patient enough to follow through on each step. How does the system need to adapt, if, for example, the only interface between the user and the system is a cell-phone? The obvious consequence of this is that no user would start with an exploration step, but rather convey her entire information need in one shot, and in speech. Apart from having to translate from speech to text, the system has to automatically determine when user feedback is needed and to minimize the interactions as much as possible.

7. CONCLUSIONS

Iterative querying is a natural paradigm for users with advanced information needs and users already do this with

search engines. Our goal in this paper has been to explore some of the issues involved in building a system supporting this paradigm in the context of *structured knowledge-bases*. We showed how this could be done with a couple of case studies and highlighted some of the tools which can already be used to implement this kind of system.

In conclusion, there is already considerable excitement over the availability of large and structured knowledge-bases. The main bottleneck is to figure out how to make use of them effectively. We believe that studying the problems of iterative querying, both at the conceptual as well as the user level will substantially advance the process of finding answers to questions (of any complexity!).

Acknowledgements. This work was partially supported by The German-Israeli Foundation for Scientific Research & Development (Grant 973–150.6/2007).

8. REFERENCES

[1] H. Achiezra, K. Golenberg, B. Kimelfeld, and Y. Sagiv. Exploratory keyword search on data graphs. In *SIGMOD*, 2010.

[2] K. Anyanwu, A. Maduko, and A. P. Sheth. Sparq2l: towards support for subgraph extraction queries in rdf databases. In *WWW*, 2007.

[3] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3), 2009.

[4] M. Cafarella. Extracting and querying a comprehensive web database. In *CIDR*, 2009.

[5] A. Doan et. al. Information extraction challenges in managing unstructured data. *SIGMOD Record*, 37(4), 2008.

[6] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, and G. Weikum. Language-model-based ranking for queries on RDF-graphs. In *CIKM*, 2009.

[7] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum. Searching rdf graphs with SPARQL and keywords. *IEEE Data Engineering Bulletin*, 33(1), 2010.

[8] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, 2008.

[9] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti. Collective annotation of wikipedia entities in web text. In *KDD*, 2009.

[10] Y. Li, H. Yang, and H. V. Jagadish. Nalix: A generic natural language search environment for xml data. *ACM Trans. Database Syst.*, 32(4), 2007.

[11] B.-C. Ooi, editor. *Special Issue on Keyword Search.* IEEE Data Eng. Bull., March 2010.

[12] J. Pound, I. Ilyas, and G. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, 2010.

[13] A. Schaefer, M. Jordan, C.-P. Klas, and N. Fuhr. Active support for query formulation in virtual digital libraries: A case study with daffodil. In *ECDL*, 2005.

DBease: Making Databases User-friendly and Easily Accessible

Guoliang Li Ju Fan Hao Wu Jiannan Wang Jianhua Feng
Department of Computer Science, Tsinghua University, Beijing 100084, China
{liguoliang, fengjh}@tsinghua.edu.cn; {fan-j07, haowu06, wjn08}@mails.thu.edu.cn

ABSTRACT

Structured query language (SQL) is a classical way to access relational databases. Although SQL is powerful to query relational databases, it is rather hard for inexperienced users to pose SQL queries, as they are required to be familiar with SQL syntax and have a thorough understanding of the underlying schema. To provide an alternative search paradigm, keyword search and form-based search are proposed, which only need users to type in keywords in single or multiple input boxes and return answers after users submit a query with *complete* keywords. However users often feel "left in the dark" when they have limited knowledge about the underlying data, and have to use a try-and-see approach for finding information. A recent trend of supporting *autocomplete* in these systems is a first step towards solving this problem. In this paper, we propose a new search method DBease to make databases user-friendly and easily accessible. DBease allows users to explore data "on the fly" as they type in keywords, even in the presence of minor errors. DBease has the following unique features. Firstly, DBease can find answers as users type in keywords in single or multiple input boxes. Secondly, DBease can tolerate errors and inconsistencies between query keywords and the data. Thirdly, DBease can suggest SQL queries based on limited query keywords. We study research challenges in this framework for large amounts of data. We have deployed several real prototypes, which have been used regularly and well accepted by users due to its friendly interface and high efficiency.

1. INTRODUCTION

Structured query language (SQL) is a database language for managing data in relational database management systems (RDBMS). SQL supports schema creation and modification, data insertion, deletion and update, and data access control. Although SQL is powerful, it has a limitation that it requires users to be familiar with the SQL syntax and have a thorough understanding of the underlying schema. Thus SQL is rather hard for inexperienced users to pose queries.

To provide an easy way to query databases, keyword search and form-based search are proposed. Keyword search only needs users to type in query keywords in a single input box and the system returns answers that contain keywords in any attributes. Although single-input-box interfaces for keyword search are easy to use, users may need an interface that allows them to specify keyword conditions more precisely. For example, a keyword may appear in different attributes, and multiple keywords may appear in the same attribute. If a user has a clear idea about the underlying semantics, it is more user-friendly to use a form-based interface with multiple input boxes to formulate queries.

However existing keyword-search-based systems and form-based systems require users to compose a *complete* query. Users often feel "left in the dark" when they have limited knowledge about the underlying data, and have to use a try-and-see approach for finding information. Many systems are introducing various features to solve this problem. One of the commonly used methods is *autocomplete*, which predicts a word or phrase that the user may type in based on the partial string the user has typed. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in partial keywords.

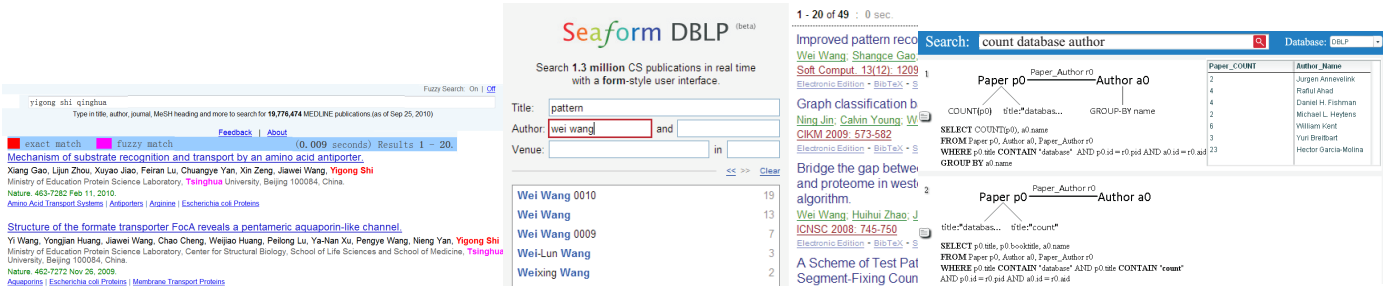
One limitation of autocomplete is that the system treats a query with multiple keywords as a single string, and it does not allow these keywords to appear at different places. For instance, consider the search box on Apple.com, which allows autocomplete search on Apple products. Although a keyword query "`itunes`" can find a record "`itunes wi-fi music store`," a query with keywords "`itunes music`" cannot find this record (as of October 2010), simply because these two keywords are not adjacent in the record.

To address this problem, recently search-as-you-type [22, 17, 16, 13, 6, 3, 1, 2, 8] is proposed in which a user types in keywords letter by letter, and the system finds answers that include these keywords (possibly at different places). For instance, if a user types in a query "`cidr database sear`" with a partial keyword "`sear`," search-as-you-type finds answers that contain complete keywords "`cidr`" and "`database`," and a keyword with the partial keyword "`sear`" as a prefix, such as "`search`." Note that the keywords may appear at different places (possibly in different attributes).

In this paper, to improve user experience of querying databases and make databases user-friendly and easily accessible, we propose a new search method, called "DBease,"[1] to improve keyword-search, form-based search, and SQL-based

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

[1] http://dbease.cs.tsinghua.edu.cn



(a) Keyword Search (PubMed) (b) Form-based Search (DBLP) (c) SQL-based Search (DBLP)

Figure 1: Screenshots of prototypes implemented using our techniques (http://dbease.cs.tsinghua.edu.cn).

search by supporting *search-as-you-type* and *tolerating minor errors* between query keywords and the underlying data. DBEASE has the following unique features. Firstly, DBEASE searches the underlying data "on the fly" as users type in keywords. Secondly, DBEASE can find relevant answers as users type in keywords in a single input box or multiple input boxes. Thirdly, DBEASE can tolerate inconsistencies between queries and the underlying data. Fourthly, DBEASE can suggest SQL queries from limited keywords.

We study research challenges in this framework for large amounts of data. The first challenge is search efficiency to meet the high interactive-speed requirement. Each keystroke from the user can invoke a query on the backend server. The total round-trip time between the client browser and the backend server includes the network delay and data-transfer time, query-execution time on the server, and the javascript-execution time on the client browser. In order to achieve an interactive speed, this total time should not exceed milliseconds (typically within 100 ms). The query-execution time on the server should be even shorter. The second challenge is to provide "on-the-fly join" for form-based search, as it is rather expensive to "join" keywords in multiple attributes. The third challenge is to infer users' query intent, including structures and aggregations, from limited keywords. To achieve a high speed for search-as-you-type, we develop novel index structures, caching techniques, search algorithms, and ranking mancinism. For effective SQL suggestion, we propose *queryable templates* to model the structures of promising SQL queries and a probabilistic model to evaluate the relevance between a template and a keyword query. We generate SQL queries from templates by matching keywords to attributes. We devise an effective ranking model and top-k algorithms to efficiently suggest the best SQL queries. We have deployed several real prototypes using our techniques, which have been used regularly and well accepted by users due to its friendly interface and high efficiency. Figure 1 gives three prototypes implemented using our techniques, which are available at http://dbease.cs.tsinghua.edu.cn.

1.1 Related Work

There have been many studies on predicting queries and user actions [19, 14, 9, 21, 20] in information search. With these techniques, a system predicts a word or a phrase the user may type in next based on the sequence of partial input the user has already typed. Many prediction and autocomplete systems[2] treat a query with multiple keywords as a single string, thus they do not allow these keywords to appear at different places in the answers. The techniques presented in this paper focus on "search on the fly," and they allow query keywords to appear at different places in the answers. As a consequence, we cannot answer a query by simply traversing a trie index (Section 2.1). Instead, the backend intersection (or "join") operation of multiple lists requires more efficient indexes and algorithms.

Bast et al. proposed techniques to support "Complete-Search," in which a user types in keywords letter by letter, and the system finds records that include these keywords (possibly at different places) [2, 3, 1, 5]. Different from CompleteSearch[3], we propose trie-based index structures and incremental search algorithms to achieve a high interactive speed. Chaudhuri et al. [6] also studied how to extend autocompletion to tolerate errors. Different from [6], we support answering multi-keyword queries.

In addition, there have been some studies on keyword search in relational databases [12, 11, 10, 18]. However they cannot support search on-the-fly.

2. IMPROVING KEYWORD SEARCH

This section improves keyword search by supporting *search-as-you-type* and *tolerating errors* between query keywords and the underlying data. We first give an example to show how search-as-you-type works for queries with multiple keywords in a relational table. Our method can be extended to support search-as-you-type on documents [13], XML data [15], and multiple relational tables [16]. Assume a relational table resides on a server. A user accesses and searches the data through a Web browser. Each keystroke that the user types invokes a query, which includes the current string the user has typed in. The browser sends the query string to the server, which computes and returns to the user the best answers ranked by their relevancy to the query. We treat every query keyword as a *partial (prefix)* keyword[3].

Formally consider a set of records R. Each record is a sequence of words (tokens). A query consists of a set of keywords $Q = \{p_1, p_2, \ldots, p_\ell\}$. The query answer is a set of records r in R such that for each query keyword p_i, record r contains a word with p_i as a prefix. For example, consider the data in Table 1, which has ten records. For a query {"vldb", "l"}, record 7 is an answer, since it contains word "vldb" and a word "luis" with a prefix "l".

Different from exact search-as-you-type, the query answer of *fuzzy* search-as-you-type is a set of records r in R such

[2]The word "autocomplete" could have different meanings. Here we use it to refer to the case where a query (possibly with multiple keywords) is treated as a *single* prefix.

[3]Clearly our techniques can be used to answer queries when only the last keyword is treated as a partial keyword, and the other keywords are treated as completed keywords.

Table 1: A set of records

ID	Record
1	EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data. Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, Lizhu Zhou. SIGMOD, 2008.
2	BLINKS: Ranked Keyword Searches on Graphs. Hao He, Haixun Wang, Jun Yang, Philip S. Yu. SIGMOD, 2007.
3	Spark: Top-k Keyword Query in Relational Databases. Yi Luo, Xuemin Lin, Wei Wang, Xiaofang Zhou. SIGMOD, 2007.
4	Finding Top-k Min-Cost Connected Trees in Databases. Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, Xuemin Lin. ICDE, 2007.
5	Effective Keyword Search in Relational Databases. Fang Liu, Clement T. Yu, Weiyi Meng, Abdur Chowdhury. SIGMOD, 2006.
6	Bidirectional Expansion for Keyword Search on Graph Databases. Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, Hrishikesh Karambelkar. VLDB, 2005.
7	Efficient IR-Style Keyword Search over Relational Databases. Vagelis Hristidis, Luis Gravano, Yannis Papakonstantinou. VLDB, 2003.
8	DISCOVER: Keyword Search in Relational Databases. Vagelis Hristidis, Yannis Papakonstantinou. VLDB, 2002.
9	DBXplorer: A System for Keyword-Based Search over Relational Databases. Sanjay Agrawal, Surajit Chaudhuri, Gautam Das. ICDE, 2002.
10	Keyword Searching and Browsing in Databases using BANKS. Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, S. Sudarshan. ICDE, 2002.

that for each query keyword p_i, record r contains a word with a prefix *similar to* p_i. In this work we use edit distance to measure the similarity between two strings. The *edit distance* between two strings s_1 and s_2, denoted by $\mathsf{ed}(s_1, s_2)$, is the minimum number of edit operations (i.e., insertion, deletion, and substitution) of single characters needed to transform the first one to the second. We say a word in a record r has a prefix w similar to the query keyword p_i if the edit distance between w and p_i is within a given threshold τ.[4] For example, suppose the edit-distance threshold $\tau = 1$. For a query {"vldb", "lvi"}, record 7 is an answer, since it contains a word "vldb" (matching the query keyword "vldb" exactly) and a word "luis" with a prefix "lui" similar to query keyword "lvi" (i.e., their edit distance is 1, which is within the threshold $\tau = 1$).

A search-as-you-type based system works as follows. The client accepts a query through the user interface, and checks whether the cached results are enough to answer the query. If not, the client sends the query to the server. The sever answers queries based on the following components. The Indexer component indexes the data as a trie structure with inverted lists on leaf nodes (Section 2.1). For each query, Searcher checks whether the query can be answered using the cached results (Section 2.2). If not, Searcher answers the query using the cache and indexes, and caches the results for answering future queries. Ranker ranks results to return the best answers (Section 2.2).

2.1 Indexer

We use a trie to index the words in the data. Each word w corresponds to a unique path from the root of the trie to a leaf node. Each node on the path has a label of a character in w. The nodes with the same parent are sorted by the node label in their alphabetical order. Each leaf node has a unique word ID for the corresponding word. The word ID is assigned in the pre-order. Each node maintains the range of word IDs in its subtree: $[minKeyID, maxKeyID]$, and the word IDs of leaf nodes under this node must be in $[minKeyID, maxKeyID]$, and vice versa. For each leaf node, we store an inverted list of record IDs that contain the corresponding word[5]. In order to improve search performance, optionally we can also maintain a forward index for the records. For each record, the forward index keeps the sorted word IDs in the record. For instance, Figure 2 shows a partial index structure for publication records in Table 1. The word "luis" has a node ID of 16. Its word ID is 7 and its inverted list includes record **7**. The word ID of node 11 is 3. The word range of node 11 is [3,5]. That is the IDs of words starting with "li" must be in [3,5]. The forward list of record 7 includes word IDs 2, 7, and 8.

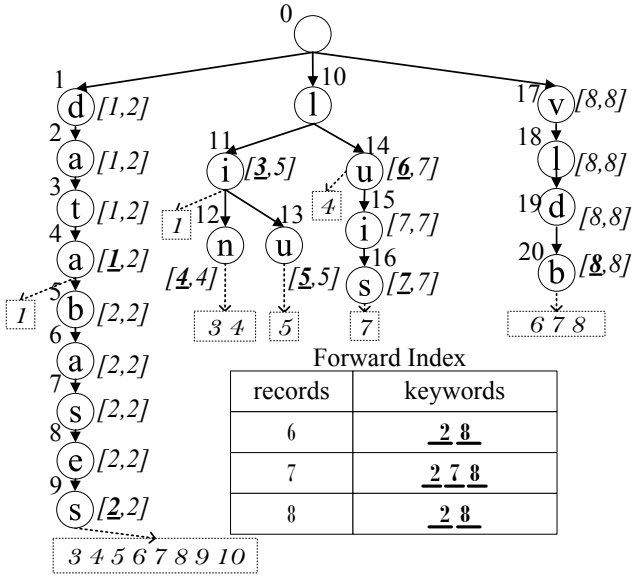


Figure 2: The trie structure and the forward index.

[4]For simplicity, we assume a threshold τ on the edit distance between similar strings is given. Our solution can be extended to the case where we want to increase the threshold τ for longer prefixes.

[5]In the literature a common "trick" to make sure each leaf node on a trie corresponds to a word and vice versa is to add a special mark to the end of each word. For simplicity we do not use this trick in the figure, and a leaf node refers to a word in the data.

2.2 Searcher

2.2.1 Exact Search

Consider a single-keyword query $c_1c_2 \dots c_x$, in which each c_j is a character. Let $p_i = c_1c_2 \dots c_i$ be a prefix query $(1 \leq i \leq x)$. Suppose n_i is the trie node corresponding to p_i. After the user types in a prefix query p_i, we store node n_i for p_i. For each keystroke the user types, for simplicity, we assume that the user types in a new character c_{x+1} at the end of the previous query string.[6] To incrementally answer the new query, we first check whether node n_x that has been kept for p_x has a child with a label of c_{x+1}. If so, we locate the leaf descendants of node n_{x+1}, and retrieve the corresponding complete words. Finally we compute the union of inverted lists of complete words using a heap-based merge algorithm. The union is called the *union list* of this keyword. Obviously the union list is exactly the answer. For instance, assuming a user has typed "`l`" and types in a character "`i`," we check whether node 10 ("`l`") has a child with label "`i`." We find node 11, retrieve complete words ("`li`, `lin`, `liu`"), and compute answers (records `1`, `3`, `4`, `5`).

It is possible that the user modifies the previous query arbitrarily, or copies and pastes a completely different string. In this case, for the new query, among all the keywords typed by the user, we identify the cached keyword that has the *longest* prefix with the new query. Formally, consider a cached query with a single keyword $c_1c_2 \dots c_x$. Suppose the user submits a new query with a single keyword $p = c_1c_2 \dots c_id_{i+1} \dots d_y$. We find $p_i = c_1c_2 \dots c_i$ that has a longest prefix with p. Then we use the node n_i of p_i to incrementally answer the new query p, by inserting the characters after the longest prefix of the new query (i.e., $d_{i+1} \dots d_y$) one by one. In particular, if there exists a cached keyword $p_i = p$, we use the cached records of p_i to directly answer the query p. If there is no such a cached keyword, we answer the query from scratch.

For a multi-keyword query, we first compute the union list of each keyword, and then intersect these union lists to compute the results. We can use two methods to compute the results. The first one is to use a merge-join algorithm to intersect the (pre-sorted) lists. Another method is to check whether each record on the shortest lists appears on other lists by doing a binary search. The latter method has been shown to achieve a higher performance in our experiments.

2.2.2 Fuzzy Search

In the case of exact search, there exists only one trie node corresponding to a partial keyword k_j. However, to support fuzzy search, there may be *multiple* prefixes similar to the keyword. We call the nodes of these similar prefixes the *active nodes* for keyword k_j. Thus for a single-keyword query, we first compute its active nodes, and then locate the leaf descendants of the active nodes. Finally we compute the *union list* of this keyword by computing union of inverted lists of all such leaf descendants. Obviously the union list is exactly the answer of this keyword. For example, consider the trie in Figure 2. Suppose $\tau = 1$ and a user types in a keyword "li." The words "li," "lin," "liu," "lu," and "lui" are all similar to the keyword, since their edit distances to "li" are within a threshold $\tau = 1$. Thus nodes 11, 12, 13, 14, and 15 are active nodes. We find the leaf descendants of the active nodes as the similar complete words ("li," "lin," "liu," "lu," and "luis"). We compute the union of inverted lists of these complete words as answers (records 1, 3, 4, 5, and 7).

Next we study how to incrementally compute active nodes for a keyword as the user types in letters. Given a keyword k_j, different from exact search which keeps only one trie node, we store a set of active nodes. We compute k_j's active-node set based on that of its prefix. The idea behind our method is to use the prefix pruning. That is, when the user types in one more letter after k_j, only the descendants of the active nodes of k_j could be active nodes of the new query, and we need not consider other trie nodes. We use this property to incrementally compute the active-node set of a new query, and refer to [13] for more details.

For a multi-keyword query, we first compute union list of each keyword, and then intersect the union lists to compute the answers. Note that as a keyword may have many active nodes and large numbers of complete words, if the sizes of these lists are large, it is computationally expensive to compute these union lists. Various algorithms can be adopted here. Specifically, we can use two methods. The first one is to use a merge-join algorithm to intersect the lists, assuming these lists are pre-sorted. Another method is to check whether each record on the short lists appears on other long lists by doing a binary search. The second method has been shown to achieve a high performance [13]. Figure 3 (a) illustrates an example in which we want to answer query "`li database vld`" using the first-union-then-intersection method.

To improve the performance, we propose a forward-index based method, which only computes the union list for a single keyword. We choose the keyword with the shortest union list, and only compute its union list. We use the forward index to check whether each candidate record on the shortest union list contains similar prefixes of other keywords. If so, this record is an answer. For each of other keywords, for the word range of each of its active node, for example $[l, u]$, we check whether the candidate record contains words in $[l, u]$. We use a binary-search method to find the word ID in the corresponding forward list, and get the smallest word ID on the list that is larger than or equal to l. Then we check whether the word ID is smaller than u. If so, this candidate contains a word in $[l, u]$, that is the record contains a prefix similar to the keyword. Thus we can use this method to compute the answer. Figure 3 (b) illustrates the forward-list-based method to answer query "`li database vld`".

2.3 Ranker

In order to compute high-quality results, we need to devise a good ranking function to find the best answers. The function should consider various factors such as the edit distance between an active node and its corresponding query keyword, the length of the query keyword, the weight of each attribute, and inverted document frequencies. If edit distance dominates the other parameters, we want to compute the answer with smaller edit distances. If there are no enough top answers with edit distance τ, we then compute answers with an edit distance $\tau + 1$, and so on. Thus, when computing the union lists, we always first compute those of the active nodes with smaller edit distances. If there are enough top answers in the intersection list of such union lists,

[6] In the general case where the user can modify the current query arbitrarily, we find the cached keyword that has the longest prefix with the input keyword, and use the same method to incrementally compute the answers.

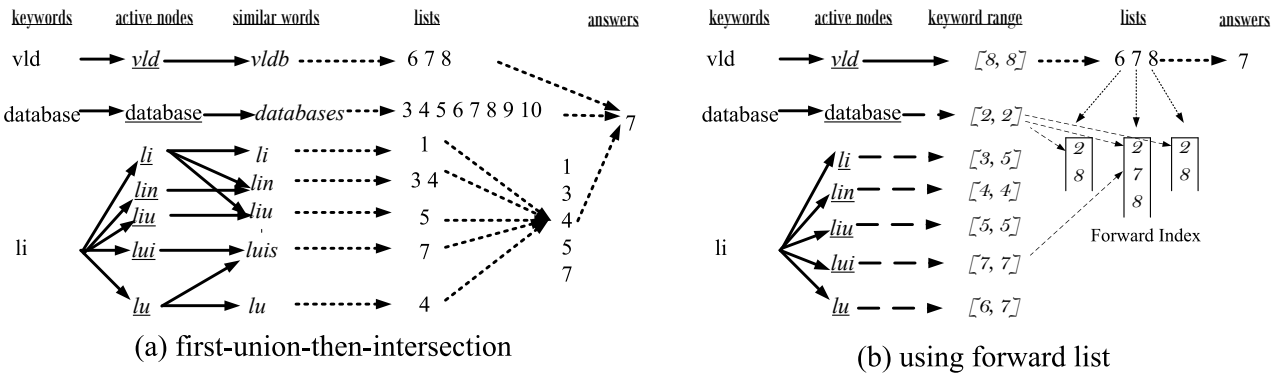


Figure 3: Two methods for answering a keyword query "`li database vld`"

we can do an early termination. We can also develop Fagin algorithms [7] to efficiently compute the top-k answers.

2.4 Additional Features

We have implemented two prototypes for keyword search on PubMed and DBLP. Figure 1(a) shows a screenshot on PubMed. In addition to the features of search-as-you-type and tolerating errors, we demonstrate the following features.

Highlighting Similar Prefix: We show how to highlight a prefix in the results that best matches a keyword. Highlighting is straightforward for the case of exact matching, since each keyword must be a prefix of the matching word. For the case of fuzzy matching, a query keyword may not be an exact prefix of a similar word. Instead, the query keyword is just similar to some prefixes of the complete word. Thus, there can be multiple similar words to highlight. For example, suppose a user types in "`lus`," and there is a similar word "`luis`." Both prefixes "`lui`" and "`luis`" are similar to "`lus`." There are several ways to highlight "`luis`," such as "**luis**" or "**luis**." We highlight the longest matched one ("**luis**").

Using Synonyms: We can utilize a-priori knowledge about synonyms to find relevant records. For example, in the domain of person names, "`William` = `Bill`" is a synonym. Suppose in the underlying data, there is a person called "`William Kropp`." If a user types in "`Bill Cropp`," we can also find this person. To this end, on the trie, the node corresponding to "`Bill`" has a link to the node corresponding to "`William`," and vise versa. When a user types in "`Bill`," in addition to retrieving the relevant records for "`Bill`," we also identify those of "`William`" following the link. In this way, our method can be easily extended to utilize synonyms.

3. IMPROVING FORM-BASED SEARCH

Note that keyword search cannot support aggregation queries, and form-based search can address this problem. This section improves form-based search by supporting *search-as-you-type* and *faceted search*. To allow users to search on different attributes, we partition the original table into several *local tables*. A local table stores the distinct values of an attribute. Each record in a local table is called a *local record*, and is assigned with a *local id*. Accordingly, the original table is called the *global table*, in which each record is called a *global record* and is assigned with a *global id*. We associate each local table with one or more input boxes in the form. For each query triggered by a keystroke in an input box, the system returns to the user not only the global ids (called the *global results*), but also the matched local ids in the corresponding local table (called the *local results*). For example, in Figure 11(b), if we type in keywords "`wei wang`" in the Author input box, the system returns the names of matched authors below the form (local results), such as `Wei Wang` and `Weixing Wang`, and their publications on the right side (global results). Next we extend the architecture for search-as-you-type to support form-based search.

3.1 Indexer

We first read the global table stored in the disk and split it into local tables. For each local table, we tokenize each record into words, and build the following index structures.

1. A trie structure with inverted lists on the leaf nodes. In the trie structure, a path from the root to a leaf corresponds to a word. The local ids for the word are added to the inverted list of the corresponding leaf node. These structures are used to efficiently retrieve the local-id lists according to query keywords.
2. A local-global mapping table. This table is used to map a local id to its corresponding global ids, so that we can retrieve the global results based on the local results. The ℓ-th row of the mapping table stores the ids of all the global records containing the ℓ-th local record. Given a set of local ids, we can obtain the corresponding global ids using this table. Take Figure 11(b) as an example. The local result "`Wei Wang`" has a local id. Its corresponding global records are the first and third publications. These two global results can be retrieved using the local-global mapping table.
3. A global-local mapping table. This table is used to map a global id to its local ids, so that we can get the local results based on the global results. The g-th row of the table stores the ids of all local records contained in the g-th global record. This table is used for the *synchronization* operations which are necessary as the local results are also affected by other attributes. For example, when the focus of input boxes is changed, we need to retrieve the correct local results of the focus based on the current

global results. For instance, in Figure 11(b), when the focus is changed to Title from Author, we need to update the local results of Title based on the global results using this mapping table.

3.2 Searcher

A query of a form-based interface can be segmented into a set of *fields*, each of which contains the query string of the corresponding input box. When a query is submitted, the system first checks whether the query can be answered from the cached results. If the query can be obtained by extending a field of a cached query with one or more letters, then we have a cache hit. We call this cached query the *base query* and cached results the *base results*. The Searcher performs an incremental search based on *base results* if there is a cache hit. Otherwise, we do a basic search as follows.

Basic search. When we cannot find cached results to answer the query, we split the query into a sequence of subqueries, in which each query appends a word to the previous query. Thus the sequence starts from an empty query and ends with the issued query. The final results can be correctly calculated if we use each of these queries one by one as the input of the *incremental search* algorithm (described below). For example, if a user inputs "`jiawei han`" in the Author input box and none prefix of the query is cached, we split it into three sub-queries, ϕ, `jiawei`, `jiawei han`. We send them one by one to the incremental-search algorithm.

Incremental search. This type of search uses previously cached results to answer a query.

Step 1. Identify the difference between the base query and the new query. We use f_i to denote the currently edited field (the i-th field), and use w to denote the newly appended keyword.

Step 2. Calculate the local ids of f_i based on the query string in f_i, by first merging the id lists of all leaf descendants of the trie node corresponding to keyword w and then intersecting the merged list with the local base results of f_i.

Step 3. Compute the global results, by first calculating the set of global ids corresponding to the local results of f_i (Step 2) using the local-global mappings and then intersecting the set with the global base results.

Step 4. Calculate the local results of f_i, called "synchronization," by first calculating the set of local ids corresponding to the global results using the global-local mapping table and then intersecting it with the local base results of f_i.

Note that we need to calculate the aggregations of each local result. Using the local-global mapping table, we can easily calculate the number of occurrences of a local result in the global result. For example, in Figure 11(b), the number "13" on the right of the entry "`Wei Wang`" means that "`Wei Wang`" appears 13 times in the global results. Next we discuss how to rank the local results and global results so as to return top-k answers. The design of the ranking function depends on the application. For example, we can rank the results according to the values of an attribute, e.g., Year or Rating. Ranking functions will be added in the final paper.

3.3 Improvements

In step 3, to obtain the global results, we map the local ids calculated in step 2 to lists of global ids, merge these lists, and then intersect the merged list with the global base results. The number of lists to be merged is equal to the number of local ids. If there are many local ids, the merge operation could be very time consuming. To address this problem, we propose another index *dual-list tries* by attaching an inverted list of global ids to each of the corresponding trie leaf nodes. In this way, given a keyword prefix, we can identify the global record that contain the keyword without any mapping operation. In addition, the number of lists to be merged is the number of complete words, which is often much smaller than the number of local ids. A smaller number of lists leads to faster merge operations. Using dual-list tries, the overall search time can be reduced compared with using original tries (called *single-list tries*).

Since we can identify the global ids using dual-list tries, those local-global mapping tables are no longer needed. In addition, we propose an alternative method to calculate the aggregations without using those local-global mapping tables. Given both local results and global results, we assign each local result a counter, initialized as 0. We first map each global id back to a list of local ids using the global-local mapping table. Then, if a local result appears in the mapped list, its corresponding counter, i.e., its occurrence number, is increased by 1. Using this method to compute the aggregations, the local-global mapping tables are not needed and we can remove them to reduce the index size.

3.4 Additional Features

We have implemented two prototypes for form-based search on DBLP and IMDB datasets. Figure 1(b) shows a screenshot of a system on the DBLP dataset. We describe the following main features of our form-based search systems.

Precise search paradigm: Suppose a user wants to find papers written by `Wei Wang` whose titles contain the word `pattern`. If she types in "`wei wang pattern`" in keyword-search system, many returned results are not very relevant. In contrast, if she types in `wei wang` and `pattern` in different input boxes in our system, she can find high-quality results.

Search-as-you-type: Suppose the user wants to find the movie titled `The Godfather` made in 1972 using the IMDB Power Search interface. She is not sure if there is a space between the word `god` and the word `father`, so she fills in the Title input box with `god father`. Unfortunately, she can not get relevant result. So she has to try several new queries. In contrast, in our system, she can modify the query and easily find the results.

Faceted search and aggregation:. Suppose a user has limited prior knowledge about the KDD conference and wants to know more about it. At first, she wants to know how many papers were published in this conference each year. She types in `kdd` in the Venue input box and then changes the editing focus to the Year input box. The listed local results show the years sorted by the number of published papers. Next, she wants to know the number of published papers of each author in KDD 2009. She chooses the year 2009 by clicking on the list, and changes the focus to the Author input box. The list below the form shows the authors, and she can see the most *active* authors. After several rounds of typing and clicking, she can get a deeper understanding about the KDD conference.

4. IMPROVING SQL-BASED SEARCH

SQL-based method is more powerful than keyword search and form-based search, and in this section we improve SQL-based search to combine the user-friendly interface of keyword search and the power of SQL. As users type in keywords, we on-the-fly suggest the top-k relevant SQL queries based on the keywords, and users can select SQL queries to retrieve the corresponding answers. For example, consider a database with tables "paper, author, write" in Table 2, where "paper" contains paper information (e.g., title), "author" contains author information (e.g., author name), and "write" contains paper-author information (e.g., which authors write which papers). Suppose an SQL programmer types in a query "`count database author`." We can suggest the following SQL queries.

1) SELECT COUNT(P.title), A.name
FROM PAPER AS P, WRITE AS W, AUTHOR AS A
WHERE P.title CONTAIN "database" AND
P.id = W.pid AND A.id = W.aid
GROUP BY A.name

2) SELECT P.title, A.name
FROM PAPER AS P, WRITE AS W, AUTHOR AS A
WHERE P.title CONTAIN "database" AND
P.title CONTAIN "count"
P.id = W.pid AND A.id = W.aid

where CONTAIN is a user-defined function (UDF) which can be implemented using an inverted index. The first SQL is to group the number of papers by authors, and the second one is to find a paper as well as its author such that the title contains the keywords "`database`" and "`count`" . We can provide graphical representation and example results to each suggested SQL as shown in Figure 1(c). The user can refine keywords as well as the suggested queries to obtain the desired results interactively.

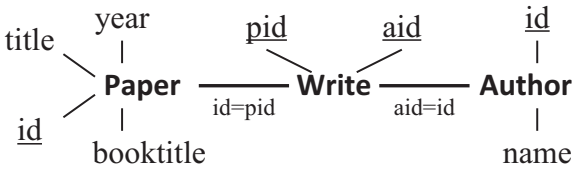
Compared with Candidate Networks (CNs) [18, 4, 12, 10] which only generate SPJ (Selection-Projection-Join) queries, our method has the following advantages. (1) We can not only suggest SPJ queries, but also support aggregate functions. (2) We can group the results by their underlying query structures, rather than mixing all results together. Our SQL-suggestion method has the following unique features. Firstly, it helps various users (e.g., administrators, SQL programmers) formulate (even complicated) structured queries based on limited keywords. It can not only reduce the burden of posing queries, but also boost SQL coding productivity significantly. Secondly, it helps users express their query intent more precisely than keyword search and form-based search, especially for complex queries. Thirdly, compared with keyword search, our method has performance superiority as we only need to first generate SQL queries and then return answers after users select an SQL query.

4.1 Overview

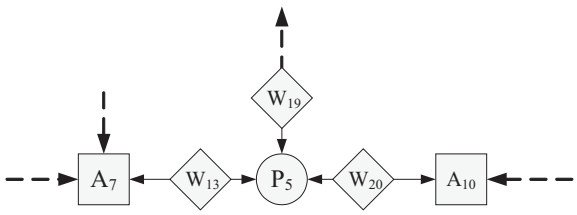
Our work focuses on suggesting SQL queries for a relational database $\mathcal{D}$ with a set of relation tables, $R_1, R_2, \ldots, R_n$, and each table R_i has a set of attributes, $A_1^i, A_2^i, \ldots, A_m^i$. To represent the schema and underlying data of $\mathcal{D}$, we define the schema graph and the data graph respectively.

To capture the foreign key to primary key relationships in the database schema, we define the *schema graph* as an undirected graph $G_S = (V_S, E_S)$ with node set V_S and edge set E_S: 1) each node is either a relation node corresponding to a relation table, or an attribute node corresponding to an attribute; 2) an edge between a relation node and an attribute node represents the membership of the attribute to the relation; 3) an edge between two relation nodes represents the foreign key to primary key relationship between the two relation tables.

Similarly, we define the *data graph* to represent the data instances in the database. The data graph is a directed graph, $G_D = (V_D, E_D)$ with node set V_D and edge set E_D, where nodes in V_D are data instances (i.e., tuples). There exists an edge from node v_1 to node v_2 if their corresponding relation tables have a foreign key (v_1) to primary key (v_2), and the foreign key of v_1 equals to the primary key of v_2.



(a) Schema graph.



(b) Data graph.

Figure 4: Schema graph and data graph [8].

Table 2 provides an example database containing a set of publication records. The database has three relation tables, PAPER, AUTHOR, and WRITE, which are respectively abbreviated to P, A and W in the rest of the paper for simplicity. Figure 4(a) shows the schema graph of the example database. In the graph, the relation PAPER has four attributes (i.e., id, title, booktitle, and year), and has an edge to another relation WRITE. Figure 4(b) shows the data graph. In this graph, an instance of PAPER (i.e., P_5) is referred by three instances of WRITE (i.e., W_{13}, W_{19}, and W_{20}).

We focus on suggesting a ranked list of SQL queries from limited keyword queries. Note that the query keywords can be very flexible that they may refer to either data instances, the meta-data (e.g., names of relation tables or attributes), or aggregate functions (e.g., the function COUNT). Formally, Given a keyword query $Q = \{k_1, k_2, \ldots, k_{|Q|}\}$, the answer of Q is a list of SQL queries, each of which contains all keywords in its clauses, e.g., the WHERE clause, the FROM clause, the SELECT clause, or the GROUP-BY clause, etc. Since there may be many SQL queries corresponding to a keyword query, we propose to rank SQL queries by their relevance to the keyword query. For example, consider the example database in Table 2 and a keyword query "`count database author`" . We can suggest two SQL queries as follows.

Table 2: An example database (Join Conditions: Paper.id = Write.pid and Author.id = Write.aid).

(a) PAPER

id	title	booktitle	year
P_1	database ir	tois	2009
P_2	xml name count	tois	2009
P_3	evaluation	database theory	2008
P_4	database ir	database theory	2008
P_5	database ir xml	ir research	2008

(b) AUTHOR

id	name
A_6	lucy
A_7	john ir
A_8	tom
A_9	jim
A_{10}	gracy

(c) WRITE

id	pid	aid	id	pid	aid
W_{11}	P_2	A_6	W_{16}	P_3	A_6
W_{12}	P_1	A_7	W_{17}	P_4	A_7
W_{13}	P_5	A_7	W_{18}	P_4	A_8
W_{14}	P_2	A_9	W_{19}	P_5	A_9
W_{15}	P_3	A_{10}	W_{20}	P_5	A_{10}

1) SELECT COUNT(P.id), A.name
FROM P, W, A
WHERE P.title CONTAIN "database" AND P.id = W.pid AND A.id = W.aid
GROUP BY A.id

2) SELECT P.title, P.booktitle, A.name
FROM P, W, A
WHERE P.title CONTAIN "database" AND P.title CONTAIN "count" AND P.id = W.pid AND A.id = W.aid

Observed from the above SQL queries, the first one is to group the number of papers with titles containing "`database`" by authors, and the second one is to find a paper as well as its author such that the title contains the keywords, "`database`" and "`count`" .

As query keywords may refer to either data instances of different relation tables, relation or attribute names, or aggregate functions, there could be large numbers of possibly-relevant SQL queries. Thus it is a challenge to suggest SQLs based on keywords. To suggest the best SQL queries, we propose a two-step method: (1) Suggest query structures, called *queryable templates* ("template" for short), ranked by their relevance to the keyword query; (2) Suggest SQL queries from each suggested template, ranked by the degree of matchings between keywords and attributes in templates.

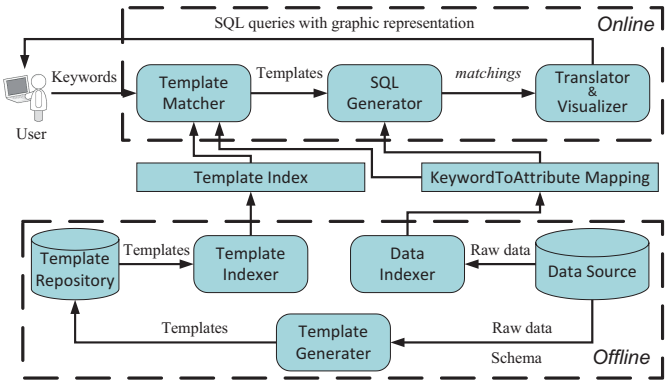
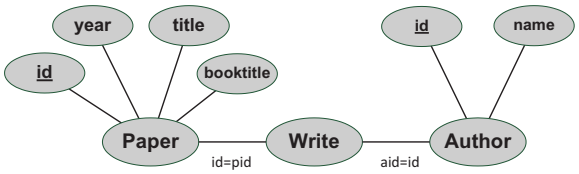
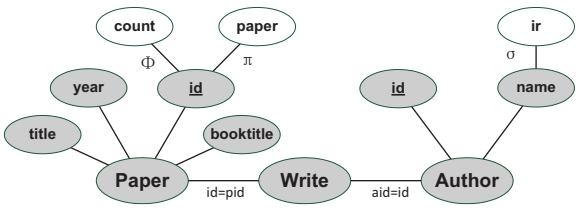


Figure 5: Architecture of SQL Suggestion [8].

Figure 5 shows the architecture of an SQL-suggestion-based system. Template Generator generates templates from Data Source and stores them in Template Repository. As there may be many templates, especially for complex schema, Template Indexer constructs Template Index for efficient online template suggestion. Since keywords may refer to data instances, meta-data, functions, etc., Data Indexer builds KeywordToAttribute Mapping for mapping keywords to attributes. Given a keyword query, Template Matcher suggests relevant templates based on the index structures. Then SQL Generator generates SQL queries from suggested templates by matching keywords to attributes. Translator & Visualizer shows SQL statements with graphical representation.



(a) A suggested template.



(b) A matching between keywords and attributes.

Figure 6: An example for query "`count paper ir`".

4.2 Template Suggestion

Template. The template is used to capture the structures of SQL queries, e.g., which entities are involved in SQL queries and how the entities are joined together. A template is an undirected graph, where nodes represent entities (e.g., `Paper`) or attributes (e.g., `year`). An edge between entities represents a foreign-primary relationship (e.g., `Paper.id=Write.pid`), and an edge between an entity and an attribute represents that the entity has the attribute. Formally, a template is defined as follows.

DEFINITION 1 (QUERYABLE TEMPLATE). *A template is an undirected graph, $G_T(V_T, E_T)$ with the node set V_T and the edge set E_T, where nodes in V_T are:*

- entity nodes: *relation tables, or*
- attribute nodes: *attributes of entities,*

and edges in E_T are:

- membership edges: *edges between an entity node and its attribute nodes, or*
- foreign-key edges: *edges between entity nodes with foreign keys and the entity nodes referred by them.*

In particular, templates with only one entity node are called ***atomic-templates****.*

For example, Figure 6(a) shows a template which involves three entities (`Paper`, `Author` and `Write`) and their attributes. Compared with CNs (trees of joined entities) [12, 11], the templates can be generated and indexed offline for fast template suggestion, and can effectively capture the relevance between keyword queries and structures.

Template Generation. Given a database, there could be a huge amount of templates that capture various query structures. We design a method to generate them using the schema graph. The basic idea of the method is to *expand* templates to generate new templates. To this end, the algorithm firstly generates atomic-templates, and takes them as bases of template generation. Then, we use an *expansion rule* to generate new templates: A template can be expanded to a new template if it has an entity node that can be connected to a new entity node via a foreign-key edge. For example, consider an atomic template, P. It can be expanded to P−W according to the expansion rule. Since a template may be expanded from more than one template, we eliminate duplicated templates. Furthermore, we examine the relationship between relation tables. For example, since the two relation tables, P and W, have a *1-to-n* relationship, i.e., an instance of W has at most one instance of P. Hence, the template, P−W−P is invalid.

Table 3: Templates for our example database (γ=5).

Size	ID	Template
1	T_1	P
	T_2	A
	T_3	W
2	T_4	P − W
	T_5	A − W
3	T_6	P−W−A
	T_7	W−P−W
	T_8	W−A−W
4	T_9	P−(W,W,W)
	T_{10}	A−(W,W,W)
	T_{11}	W−P−W−A
	T_{12}	W−A−W−P
5	T_{13}	P−W−A−W−P
	T_{14}	A−W−P−W−A
	...	...

Apparently, the above-mentioned expansion rule may lead to a combinatorial explosion of templates. To address this problem, we employ a parameter γ to restrict the maximal size of templates (i.e., the number of entities in a template), which is also used in CN-based keyword-search approaches [12, 11]. The motivation here is that templates with too many entities are meaningless, since the corresponding query structures are not of interests to users. Table 3 provides the templates generated for our example database under $\gamma = 5$, where P−(W,W,W) represents that P is connected with three W entities. Even if we restrict the maximal size of templates, there still are many templates due to complex relationships between tables. We can devise an efficient top-k template ranking algorithm to avoid exploring the entire space of searching templates.

Template Suggestion Model. We propose a scoring function to measure the relevance between keywords and templates, which considers two factors: 1) the relevance between a keyword q and the entity R in a template T, denoted by $Pr(q, R)$, and 2) the importance of the entity R, denoted by $I(R)$. $Pr(q, R)$ can be taken as the probability that the entity R contains the keyword q. We use the relative frequency that R contains q to estimate this probability. We treat the entity R as a *virtual* document and estimate the likelihood of sampling keyword q from the "document" using term frequency and inverse document frequency. In particular, the virtual document R does not only have words in tuples, but also contains words in meta-data (e.g., attribute names, entity names, etc.). We use the number of tuples to estimate $Pr(q, R)$, if q refers to the meta-data of R. The importance of an entity R in a template T, $I(R)$, is computed using a graph model. We compute the *PageRank* of tuples on the data graph [18], and take the average value as template importance. Given a query Q and a template T, we combine the two factors to evaluate its score $S_1(Q, T)$:

$$S_1(Q, T) = \sum_{q \in Q} \sum_{R \in T} I(R) \cdot Pr(q, R). \quad (1)$$

Top-k Ranking Algorithm. As the number of templates could be very large for complex schemas, it is expensive to enumerate the templates. A straightforward way to address this problem is to calculate the ranking score for every template according to our template ranking model. Unfortunately, since the number of templates is exponential, this approach becomes impractical for real-world databases. Therefore, an efficient top-k ranking algorithm is rather necessary to avoid exploring all possible templates. To address this problem, we devise a threshold algorithm (TA) [7] to compute top-k templates efficiently.

Our basic idea is to scan multiple lists that present different rankings of templates for an entity, and aggregate scores of multiple lists to obtain scores for each template ($Pr(q, R)$). For early termination, the algorithm maintains an upper bound for the scores of all unscanned templates. If there are k scanned templates whose scores are larger than the upper bound, the top-k templates have been found. Interested readers are referred to [8] for more details.

4.3 SQL Generation from Templates

Given a query and a template, to generate SQLs, we need to map each keyword to an attribute in the template. This section proposes an SQL-suggestion model to suggest SQLs.

SQL Generation Model. To map keywords to a template, we construct a set of keyword-to-attribute mappings between keywords and attributes. A *matching* is a set of mappings which covers all keywords. Then we evaluate the score of each matching, find the best matching with the highest score, and generate SQL queries based on the matching.

SQL Ranking. We measure the score of each matching by considering two factors: 1) the degree of each keyword-to-attribute mapping in the matching; and 2) the importance of the mapped attributes. The degree of a mapping from a keyword q to an attribute A with a type t, denoted as $\rho_t(q, A)$, is used to determine whether q is relevant to A. We consider three types of keyword-to-attribute mappings, i.e., 1) the *selection* type (denoted as σ): q refers to data instances of A; 2) the *projection* type (denoted as π): q refers to the name of A; 3) the *aggregation* type (denoted as ϕ): q refers to an aggregate function of A. We use the relative frequency that A contains q to estimate $\rho_\sigma(q, A)$ and $\rho_\pi(q, A)$. $\rho_\phi(q, A) = 1$ if q is a function name (e.g.,

COUNT, MAX).

We employ the *entropy* to compute the importance of an attribute A, $I(A)$. Let $V = \{v_1, v_2, \ldots, v_x\}$ denote distinct values of A and let f_i denote the relative frequency that A has value v_i, and $I(A) = -\sum_{i=1}^{x} f_i \cdot \log f_i$.

Combining the two factors, we present the scoring function as follows. Consider a keyword query $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ and a set of attributes $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ in a suggested template T. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_{|\mathcal{M}|}\}$ be a matching between Q and T, where $M \in \mathcal{M}$ is a keyword-to-attribute mapping of keyword q_M and attribute A_M. The score is

$$S_2(\mathcal{M}) = \sum_{M \in \mathcal{M}} I(A_M) \cdot \rho_t(q_M, A_M). \quad (2)$$

Thus we first find the best matching (i.e., an SQL query) from a template, which can be formulated as a *weighted set-covering problem* and can be solved using a polynomial-time greedy approximation algorithm. Then, we extend the algorithm to generate top-k SQL queries by finding k best matchings with highest scores in a greedy manner. Interested readers are referred to [8] for more details.

In summary, given a keyword query, we first find relevant templates for the keywords using the threshold-based algorithms. Then for each template, we generate corresponding SQL queries using the keyword-to-attribute mappings.

4.4 Additional Features

We have implemented two prototypes for SQL-based search on DBLP and DBLife datasets. For example, Figure 1(c) shows a screenshot of a system on the DBLP dataset. We describe the main features as follows.

Predicting the query intent from limited keywords: Unlike existing SQL assistant tools and keyword-search methods, DBEASE focuses on *predicting* the query intent from keywords and *translating* them into structured queries. Suppose a user wants to find a paper with *title* containing "`data`" and *booktitle* containing "`icde`" which is written by an author "`wang`." After she issues a keyword query "`wang data icde`" to the system, the system returns SQL queries, user-friendly representation, and sample answers instantly. This search paradigm is more expressive and powerful than the conventional keyword search.

Assisting users to formulate SQL queries efficiently: To reduce the burden of writing SQL queries without the loss of expressiveness, we propose an effective ranking mechanism to suggest the most relevant SQL queries to users. Suppose that a user issues a query "`count person stanford`," she would be presented with the following SQL queries: (*i*) Find the number of people whose organization is `Stanford`; (*ii*) Find the number of people who give talks to `Stanford`; (*iii*) Find the number of people who are related to `Stanford`. The above SQL queries are useful for users to navigate the underlying data, and can reduce the burden of posing queries from several complex SQL statements to 3 keywords.

Improving the effectiveness of data browsing: Users can browse the data by typing keywords and selecting suggested SQL queries. In our systems, the answers are naturally organized based on their corresponding SQL queries. Each SQL query represents a group of records with the same structure, which could help users find similar and relevant records in one group and browse other potentially relevant records in other groups.

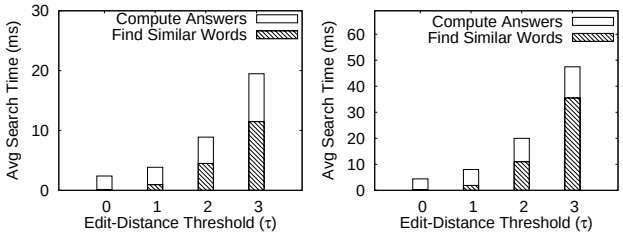
5. EXPERIMENTAL RESULTS

We have implemented our techniques on several real datasets, PubMed[7], DBLP[8], IMDB[9], and DBLife[10].

All the codes were implemented in C++. We conducted the experiment on a Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz CPU and 4 GB RAM.

5.1 Improving Keyword Search

We evaluated the performance of search-as-you-type by varying the edit-distance threshold τ. We used two datasets: DBLP and PubMed. We selected 1000 queries from query logs from our deployed systems. We implemented our best algorithms and computed the answers in two steps: (1) computing similar prefixes, and (2) computing answers based on similar prefixes. Figure 7 shows the results. Our methods could answer a query within 50 ms. The variant of our method was about 8 ms.



(a) DBLP (b) MEDLINE

Figure 7: Efficiency of search-as-you-type.

5.2 Improving Form-based Search

We evaluated the performance of form-based search. We used two datasets: DBLP and IMDB. We used a workload of 40 thousand queries collected from our deployed system. Figure 8 shows the comparison of average search time per query of four algorithms: (1) SL-BF, which uses Single-List tries and Brute-Force synchronization (do synchronization for each keystroke), (2) SL-OD, which uses Single-List tries and On-Demand synchronization (do synchronization when search focus is changed), (3) DL-BF, which uses Dual-List tries and Brute-Force synchronization, and (4) DL-OD, which uses Dual-List tries and On-Demand synchronization.

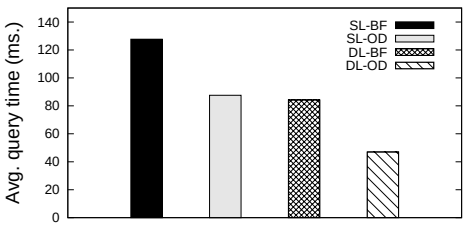


Figure 8: Efficiency of form-based search (IMDB).

We can see that both the dual-list tries and on-demand synchronization can improve the search speed. If we use these two together, the DL-OD algorithm can answer a query within 50 milliseconds. Moreover, compared with search-as-you-type, this method supports on-the-fly faceted search.

[7] http://www.ncbi.nlm.nih.gov/pubmed
[8] http://dblp.uni-trier.de/xml/
[9] http://www.imdb.com/interfaces
[10] http://dblife.cs.wisc.edu/

5.3 Improving SQL-based Search

We evaluated effectiveness and efficiency of our method (SQLSugg) and compared with Discover-II [11]. We used two datasets: DBLP and DBLife. We selected ten queries for each dataset [8]. We evaluated the precision of template suggestion (Figure 9(a)) and that of returned answers (Figure 9(b)) on the DBLife dataset. We see that our method outperforms Discover-II significantly. For example, the precision of our method is much better than that of Discover-II. The improvement of our method is due to the following reasons. Firstly, compared with Discover-II, our method allows users to search the meta-data (i.e., names of relation tables, or attributes), while Discover-II only supports full-text search. Secondly, we group the answers based on structures and employ a more effective ranking function.

We examined the efficiency of our SQL suggestion methods, and compared the query time with Discover-II. Figure 9(c) shows the experiment results on the DBLife dataset. The results show that our methods outperform Discover-II significantly. For example, consider the query time for a keyword query with length 6 in Figure 9(c). Our method outperforms Discover-II by an order of magnitude. Moreover, the query time of our method is very stable, and is always smaller than 100 milliseconds. It indicates that our method can suggest SQL queries in real time.

The improvement of our methods is mainly attributed to the top-k template ranking algorithm. Discover-II exploits a keyword-to-attribute index to find tuple sets that may contain query keywords, and on-the-fly generates candidate networks. Since the amount of candidate networks could be large, especially for complex schemas, Discover-II is inefficient to generate all candidate networks. For example, the query time of Discover-II is hundreds of milliseconds on the DBLife dataset with 14 tables. In contrast, our method focuses on suggesting top-k templates according to our ranking model. The result shows that the algorithm is very efficient and can suggest template in real time.

5.4 Scalability

We tested the scalability of our methods for different search paradigm. Figure 10 shows the scalability of efficiency of our methods. We see the search time increased linearly as the dataset increased. All average search time is less than 60 milliseconds and the variant is about 10 milliseconds. The index size also increased linearly as the dataset increased, as shown in Figure 11.

6. CONCLUSION

In this paper, we have studied a new search method DBease to make databases user-friendly and easily accessible. We developed various techniques to improve keyword search, form-based search, and SQL-based search for enhancing user experiences. Search-as-you-type can help users on-the-fly explore the underlying data. Form-based search can provide on-the-fly faceted search. SQL suggestion can help various users to formulate SQLs based on limited keywords.

We believe this study on making databases user-friendly and easily accessible opens many new interesting and challenging problems that need further research investigation. Here we give some new open research problems.

Supporting Ranking Queries Efficiently: Our proposed techniques need to first compute all candidates and then rank them to return the top-k answers. If there are larger numbers of answers, these methods are expensive. To address this problem, it calls for new techniques to support ranking queries efficiently. Different from traditional search paradigm, in search-as-you-type, there are multiple corresponding complete keywords and inverted lists for each prefix keyword. Existing threshold-based methods [7] need to scan the elements in each inverted list to compute the top-k answers and they are expensive if there are large numbers of inverted lists. We have an observation that some inverted lists can be pruned if the corresponding complete keywords are not very relevant. Especially, we can prune the inverted lists of complete keywords with larger edit distances. Thus we have an opportunity to prune some inverted lists and thus improve the search performance for ranking queries.

Supporting Non-string Data Types: Our proposed techniques only support string data types and do not support other data types, such as Integer and Time. Consider the case where we have a publication database, and a user types in a keyword "2001". Based on edit distance, we may find a record with a keyword "2011". On the other hand, if we knew this keyword is about the year of a publication, then we may relax the condition in a different way with edit distance, e.g., by finding publications in a year between 1999 and 2002. This kind of relaxation and matching depends on the data type and semantics of the corresponding attribute, and requires new techniques to do indexing and searching. Similarly, to support approximate search, we also want to support other similarity functions, such as Jaccard.

Supporting Personalized Search: Different users may have different search intentions, and it is challenging to support personalized search for various users. We can utilize user query logs and mine search interests for different users so as to support personalized search.

Client Caching: We can not only support server caching, but also use client caching to improve search performance. We can cache results of users' previous queries and use the cached results to answer subsequent queries. However it is very challenging to support client cache, since it is hard to predicate subsequent queries of users. We need to study how to cache previous queries and corresponding results.

7. ACKNOWLEDGEMENT

This work is partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2009AA011906, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and National S&T Major Project of China.

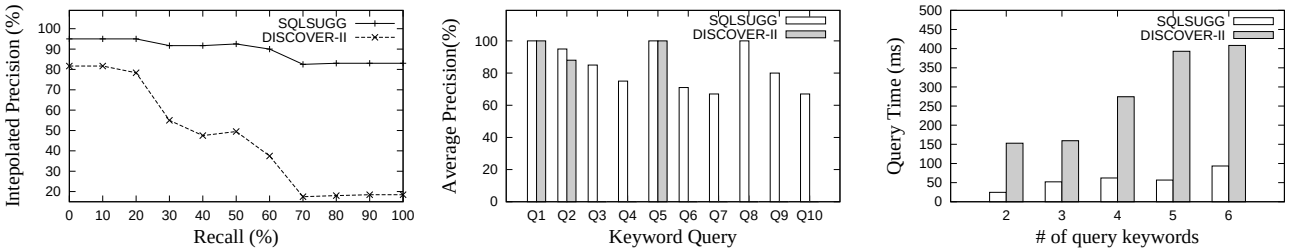
8. REFERENCES

[1] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.

[2] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

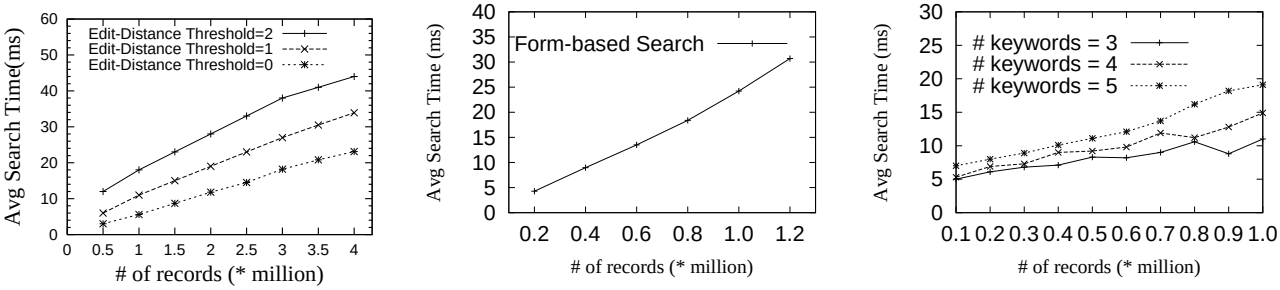
[3] H. Bast and I. Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*, 2007.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.



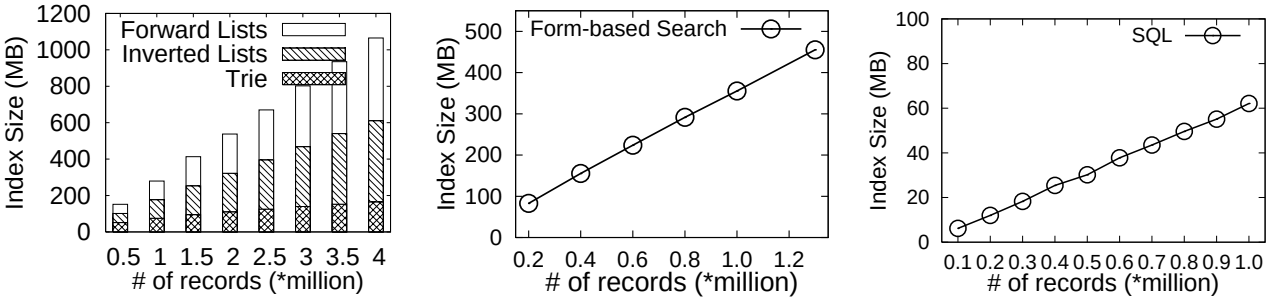
(a) Effectiveness of template suggestion. (b) Effectiveness of record retrieval. (c) Efficiency comparison.

Figure 9: Evaluation of SQL-based Search on the DBLife data set.



(a) Keyword Search (PubMed) (b) Form-based Search (DBLP) (c) SQL-based Search (DBLP)

Figure 10: Scalability of search performance.



(a) Keyword Search (PubMed) (b) Form-based Search (DBLP) (c) SQL-based Search (DBLP)

Figure 11: Scalability of index size.

[5] M. Celikik and H. Bast. Fast error-tolerant search on very large texts. In *SAC*, pages 1724–1731, 2009.

[6] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[8] J. Fan, G. Li, and L. Zhou. Interactive sql query suggestion: Making databases user-friendly. In *ICDE*, 2011.

[9] K. Grabski and T. Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, 2007.

[11] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[12] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[13] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[14] K. Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[15] G. Li, J. Feng, and L. Zhou. Interactive search in xml data. In *WWW*, pages 1063–1064, 2009.

[16] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.

[17] G. Li, S. Ji, C. Li, J. Wang, and J. Feng. Efficient fuzzy type-ahead search in tastier. In *ICDE*, pages 1105–1108, 2010.

[18] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD Conference*, pages 903–914, 2008.

[19] H. Motoda and K. Yoshida. Machine learning techniques to make computers easier to use. *Artif. Intell.*, 103(1-2):295–321, 1998.

[20] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.

[21] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[22] H. Wu, G. Li, C. Li, and L. Zhou. Seaform: Search-as-you-type in forms. *PVLDB*, 3(2):1565–1568, 2010.

Here are my Data Files. Here are my Queries. Where are my Results?

Stratos Idreos[†] Ioannis Alagiannis[⋆] Ryan Johnson[‡] Anastasia Ailamaki[⋆]

[†]CWI, Amsterdam [‡]Carnegie Mellon University [⋆]École Polytechnique Fédérale de Lausanne

ABSTRACT

Database management systems (DBMS) provide incredible flexibility and performance when it comes to query processing, scalability and accuracy. To fully exploit DBMS features, however, the user must *define* a schema, *load* the data, *tune* the system for the expected workload, and answer several questions. Should the database use a column-store, a row-store or some hybrid format? What indices should be created? All these questions make for a formidable and time-consuming hurdle, often deterring new applications or imposing high cost to existing ones. A characteristic example is that of scientific databases with huge data sets. The prohibitive initialization cost and complexity still forces scientists to rely on "ancient" tools for their data management tasks, delaying scientific understanding and progress.

Users and applications collect their data in flat files, which have traditionally been considered to be "outside" a DBMS. A DBMS wants control: always bring all data "inside", replicate it and format it in its own "secret" way. The problem has been recognized and current efforts extend existing systems with abilities such as reading information from flat files and gracefully incorporating it into the processing engine. This paper proposes a new generation of systems where the only requirement from the user is *a link to the raw data files*. Queries can then immediately be fired without preparation steps in between. Internally and in an abstract way, the system takes care of selectively, adaptively and incrementally providing the proper environment given the queries at hand. Only part of the data is loaded at any given time and it is being stored and accessed in the format suitable for the current workload.

1. INTRODUCTION

Database systems can only achieve excellent query response times given a number of low-level *designing and tuning* steps, which are a prerequisite for the system to start processing queries. A significant part of the initialization cost is due to loading the data into files or raw storage according to a format specified by the DBMS, and to physically designing and tuning the data set for the expected workload, i.e., creating the proper indices, materialized views, etc. Flat files of raw data are considered "outside" the DBMS, within which data is viewed and massaged in specific formats. Despite that the assumption of control over the data placement and format entails design advantages for the database system, such a restriction prevents the wide adoption of DBMS technology to new application areas, increases the bootstrap time of a new application and leads to systems that either are appropriate for only one scenario or need to be perennially re-tuned.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

1.1 Accessing Flat Files

Both the research community and the commercial world recognize the problem and several crucial steps have already been addressed. For example, many commercial and open-source database systems already offer a functionality that allows a database system to immediately query a flat file. The DBMS essentially links a given table of a schema with a flat file and, during query processing, parses data from the flat file on-the-fly. Oracle, for instance, offers an option to have "external tables" while MySQL enables the "CSV engine". As a result, data can be queried without having to explicitly load the raw data into the DBMS. In practice, however, flat files are still "outside" the DBMS as there is no support for indices, materialized views or any other advanced DBMS optimization. Query processing performance is therefore lower when compared to the the performance of queries running on "internal" tables, so the systems mostly offer external flat files as an alternative way for the user to load/copy data into normal DBMS tables rather than for query processing. Consequently, even though current flat file functionality is a significant step forward, the problem is still an open one as there is still a long way to go towards systems that require zero initialization overhead.

1.2 Motivating Example

Typically, scientific databases grow on a daily basis as the various observation instruments continuously create new data. Thus, the optimal storage and access patterns may change even on a daily basis purely depending on the new data, its properties, correlations, as well as the ways that the scientists navigate through the data and the ways their understanding and data interpretation evolve. In such scenarios, no up-front physical design decision can be optimal in light of a completely dynamic and ever-evolving access pattern. Therefore, people often rely on Unix-based tools or on custom solutions in order to achieve bread-and-butter

functionality of a DBMS.

Paying a heavy preparation cost for the unknown is prohibitive if needed on a daily basis. Even simply loading the data in the database is a significant investment and delay for large data sets. Alternatively, one can write a quick Awk script and immediately query any part of the data, avoiding all hassle associated with designing schema, loading data and tuning. Here follow a few example reactions from scientists:

1. Why do I have to wait multiple hours for loading and tuning? I am getting another Terabyte of data tomorrow and I just want to quickly find out if the current data is of any interest so that I go ahead and analyze it.

2. Why should I have to load the whole data set? I just need a small part now; I do not know if, or when, I will need the rest.

3. How can I decide what indices to build or whether to use column- or row-stores? I won't understand the data properties until after I've looked at it! Worse, tomorrow could be a different story.

4. How should I know how to set-up a complex DBMS? I am not a computer scientist. Hiring DB experts to set-up and re-tune the system on a daily basis is extremely expensive.

1.3 Contributions

In this paper, we argue towards a new generation of systems that provide a *hybrid* experience between using a Unix tool and a DBMS; all they need as a start-up step is *a pointer to the raw data files*, i.e., the flat files containing the data, for example in CSV format. There is *zero initialization* overhead as with a scripting tool but at the same time *all advanced* query processing capabilities and performance of a DBMS are retained. The key idea is that data remains in flat files. The user can edit or change a file at any time. When queries arrive, the system will take care of bringing the proper data from the file, and it will store it and evaluate it in an appropriate way.

We first demonstrate the initialization overhead of DBMS and the flexibility of using a scripting language. Then, we demonstrate the suitability of a DBMS for data exploration, when a quick glimpse over the data is not the only goal, i.e., when a user wants to repeatedly query the same data parts. Finally, we describe our vision in detail. Various policies are studied regarding how data can be fetched, cached, reused and how this whole procedure can happen on-the-fly, integrated with query processing in a modern DBMS. We provide a prototype implementation and evaluation over MonetDB. Our results clearly show the potential and benefits of the approach as well as the opportunity to further study and explore this topic.

2. TO DB OR NOT TO DB?

This section provides more concrete motivation of why using DBMS can be of significant importance in new scenarios like scientific databases, as well as of the fact that drastic changes are needed, and ends by discussing our vision and research challenges.

As a starting point we perform a study of how using a DBMS compares against using Unix tools for query processing. We compare the popular and powerful Awk scripting language with a state of the art open-source column-store DBMS, MonetDB. We use a 2.4 GHz Intel Core2 Quad CPU equipped with one 32 KB L1 cache per core, two 4 MB L2 caches, each shared by 2 cores, and 8 GB RAM and two 500 GB 7200 rpm SATA hard disks configured as software-RAID-0. The operating system is Fedora 12.

The set-up is as follows. The data set consists of a four-attribute table, which has as values unique integers randomly distributed in the columns. The queries are always 10% selective, and follow the template below (Q1):

```
    select sum(a1),min(a4),max(a3),avg(a2)
Q1  from R
    where a1>v1 and a1<v2 and a2>v3 and a2<v4
```

Figure 1 shows the results for various different input sizes and two kinds of costs: the loading cost and the query processing cost. For Awk there is no loading cost, while for the DBMS, the complete loading cost has to be incurred before we can process any data. For the query processing performance of the DBMS, we report both hot and cold runs, as well as evaluation over an adaptive indexing scheme. In the next two subsections, we study these results from two perspectives; In the first part, we discuss clear disadvantages for the DBMS, while in the second part we discuss clear DBMS advantages. The third subsection then introduces our vision for a hybrid system.

2.1 Why Not Databases

Let us first discuss the disadvantages of using a DBMS.

Loading Overhead. With the term "loading" we refer to the procedure of copying the data from the flat files into the DBMS. Our flat files are in CSV format. To process even a single query in a DBMS we first have to incur the *complete* loading cost. Thus, the first query response time can be seen as the cost to load the data plus the cost to run the actual query. Figure 1 shows that the loading cost of the DBMS represents *a significant bottleneck* when it comes to large data sizes. In other words, loading the data and evaluating a query with the DBMS is much slower than simply firing an Awk script. If we add the loading cost and the query processing cost of the DBMS for the case of 1 Billion tuples, it is 800 seconds higher than that of running an Awk script. And of course, here we ignore the expert knowledge and time required to set-up the DBMS (e.g., with the proper physical design).

In addition, while the query processing performance of Awk scales perfectly with input sizes, this is far from true for the loading cost of the DBMS. Recall that the loading cost is to be considered as part of the first query in a DBMS. What actually happens is that for the smaller sizes everything fits quite comfortably in memory and is never written to disk during the experiment. For the input of the 1 Billion tuples table, however, the system reaches the memory limits and needs to write the table back to disk; exactly then, we pay the significant cost of the loading procedure.

Not a "Quick Look" to the Data. This experiment represents the *ideal* scenario for a DBMS, i.e., the query is interested in all 4 attributes of the table. This way, even for the case of 1 Billion tuples where loading is a significant

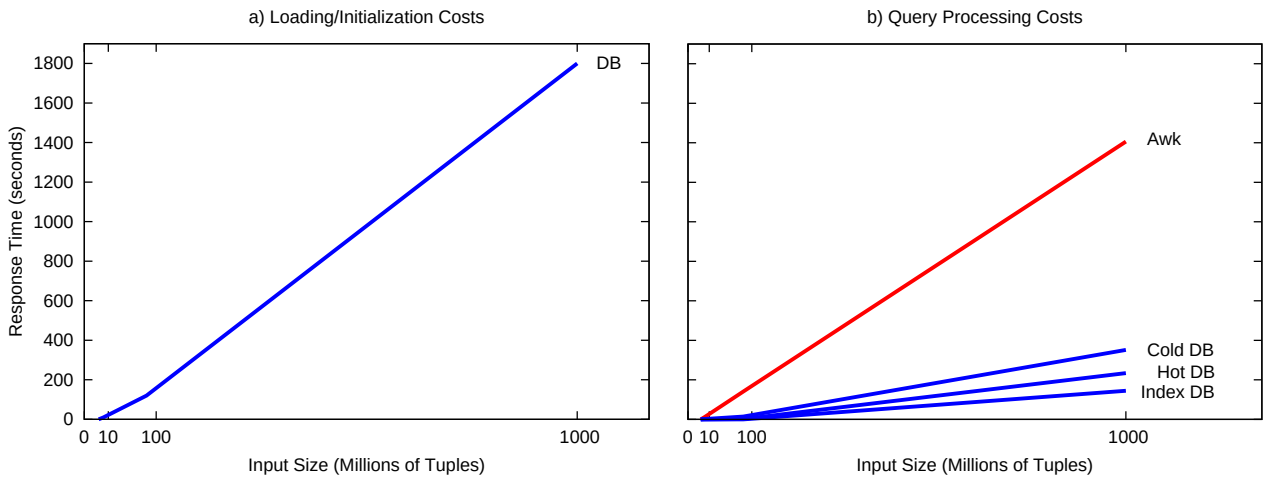


Figure 1: DB Vs. Unix tools

cost, the DBMS did not do any extra work, i.e., it did not load any unnecessary columns.

In scenarios like scientific databases though, it is typical to have hundreds or even thousands of columns, while a query might be interested in only a few of those. A DBMS has to first load all columns before it can process any queries. It has to *own and replicate* the complete data set. On the contrary, a scripting tool like Awk can provide a quick and "painless" gateway to the data. We can explicitly identify the data parts we are interested in, while zero preparation or pre-processing is required. We do not need to replicate the data and we can actually edit the data with a text editor directly at any time and fire a query again.

The above observations make scripting languages like Awk a much more flexible and useful tool than a DBMS for inspecting and quickly initiating explorations over large data sets.

2.2 Why Databases

The previous section argues that Awk is great for a quick glimpse of the data. The DBMS is not. For repeated queries over the same data, however, Awk lacks the smarts to improve performance; this is where the DBMS shines! Here, we discuss these DBMS benefits.

Performance Gains After Loading. Let us ignore the loading costs and concentrate purely on the query processing costs in Figure 1. For all sizes, we see a consistent pattern with the DBMS being significantly faster than the pure performance the optimized Awk script can provide. No matter if we look at cold or hot queries the DBMS is much stronger. The indexed DB curve indicates an even better performance using an optimized physical design, where storage is adapted to the queries via index creation (in this case selections and tuple reconstruction are significantly improved using database cracking [12]). Especially as the data grows, the DBMS is one order of magnitude faster. By having the data loaded and stored in a proper format, the DBMS does not have to go through an expensive tokenization and parsing procedure again and again; it only pays this cost during loading. Consequently, even the non-indexed DBMS runs can materialize significant benefits over Awk that always has to go through the flat files.

Exploratory Behavior Benefits. Advanced data management scenarios like the analysis of scientific data is far from an one query task. It typically involves a lengthy sequence of queries which dynamically adapts based on how the scientist interprets the data, continuously zooming in and out of data areas representing an exploratory behavior. A scripting tool has a constant performance that cannot improve over time. A DBMS, on the other hand, has an one-time up-front cost and from there on performance is very good making it a great fit for when we know we are going to query again and again the same data parts. In addition, indices, optimizers and advanced operator implementations guarantee near-optimal usage and exploitation of hardware and workload properties. A DBMS can be tuned to properly scale to hundreds of nodes or CPUs, it can be tuned to adapt to skew of data or queries, etc. On the other hand, scripts typically are single threaded processes without the notion of optimization and adaptation.

Fast Evaluation of Complex Queries. Experiments with more complex queries show even bigger benefits. For example a join query with a few aggregations on two 10^8 tuples tables (1 to 1 join) takes 387 seconds on a hash join implementation in Awk, while it takes 247 seconds if we sort the data (using the Unix sort tool) and then implement a merge join in Awk (a 100 lines script). On the contrary, a cold DB run takes 39 seconds while a hot one only 5!

Declarative SQL Interface. Expressing queries in the declarative SQL language is a major benefit of a DBMS for flexibility and query reuse/editing. On the contrary, with Awk one has to be an expert in scripts. In addition, workload knowledge is needed for optimal performance. Our scripts "match" the techniques used in an optimized DB plan, i.e., push down selections, perform the most selective filtering first, etc. Even though using Awk in everyday scripting is an invaluable tool, the overhead of writing complex scripts to match SQL expressiveness should not be underestimated. Our experience shows that, even for an expert, it requires several hours to produce efficient scripts. A simple 1-2 line SQL query needs several tenths or hundreds of lines in a scripting language. Thus, the even harder part becomes that of maintenance and reuse. In our inter-

action with scientists in EPFL, several of them admit that they have numerous scripts that they either never attempt to edit or they do not remember any more what they do.

We repeat the above experiments using also Perl instead of Awk. This was two times slower than the Awk scripts. Another heavily used tool is Matlab. It specializes in scenarios where heavy computation is needed. We did not try it out in our experiments but nevertheless the argumentation remains the same as with all scripting and low level languages. Similarly, one may actually write pure C code which is expected to be faster for targeted queries than a generic DBMS.

None of the methods above provides the flexibility and scalability of a DBMS and of course they require expert technical knowledge as well as a good understanding of the data. DBMS were built based upon this motivation; *generic, abstract and declarative* usage of an information management system that can actually perform very well. The users need to care only about what they want to obtain from the system and not about how to obtain it.

2.3 Vision & Challenges

The previous sections showed both major advantages and disadvantages for using a DBMS. Here, we discuss our vision towards a hybrid system that blends the best of scripting tools and DBMSs.

2.3.1 The Vision

The ultimate goal is a system that avoids all the initialization steps and hassles of a DBMS, but at the same time it maintains and even enhances the potential performance gains.

All you need to do to use it, is point to your data and you can start querying immediately with SQL queries.

Without any external administration or control and without any preparation steps, the system is immediately ready to answer queries. It selectively brings data from the flat files into the database during query processing. It adaptively stores these data into the proper format, adjusting its access methods at the same time, all driven by the current hot workload needs. Column-store, row-store and hybrid storage and execution patterns all co-exist even for the same or overlapping parts of the data set.

2.3.2 Challenges

The challenge is to make all this continuous and ever evolving process as transparent as possible to the user via lightweight adaptation actions and rapid response times. Conceptually we strive to blend the immediate and interactive feeling and simplicity of using Unix-like tools to explore data with the flexibility, performance and scalability of using a DBMS. Some of the main research questions we have to answer are the following.

1. When and how to load which parts of the data?
2. How to store each data part we load?
3. How to access each data part?

The first one represents a completely new problem. Up to now in order to use a DBMS we need to completely load the data. There is no notion of partial, incremental and adaptive loading.

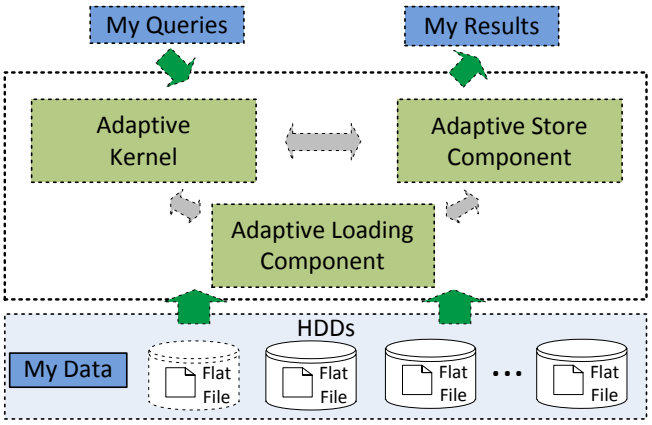


Figure 2: System Architecture

For questions 2 and 3 the database community has done extensive research to find the optimal storage and access patterns for *specific* workloads and scenarios. Nevertheless, there is no notion of an adaptive, ever changing, dynamic storage *and execution* kernel.

2.3.3 Basic Components

The above goals open a broad research landscape, which touches every corner of core database systems design. Loading, storage and execution, they all need to adopt a self-organizing nature, capable of adapting to the workload, requiring zero or minimum human effort. The key to our research path is that:

Queries become the first class citizen that define loading, storage and execution patterns and strategies.

The components of the envisioned architecture as well as the ways that these components interact can be seen in Figure 2.

An *adaptive loading* component will be responsible to always make sure that just enough data is loaded but also that we can easily access the rest of the data if needed.

An *adaptive store* component will make sure that data is stored in ways that fit the workload.

In the same philosophy, an *adaptive kernel* will at any time contain multiple different execution strategies to best suit the observed workload.

In the rest of this paper, in Sections 3 and 4, we provide an initial study and an implementation prototype of adaptive loading. This is the first critical component needed to realize this vision. In addition, it probably represents the most provocative of all challenges given that database systems always like to fully *control and manage* the data in their own format and environment as opposed to relying on flat files. Section 5 sketches the research landscape. It discusses in more detail the challenges and opportunities associated with the adaptive store and the adaptive kernel concepts. We also provide an extensive discussion of how several core database design problems need to be rethought in this vision, e.g., issues that have to do with updates, concurrency control and robustness.

3. ADAPTIVE PARTIAL LOADING

In this section, we study adaptive loading in more detail, providing a series of techniques as well as an implementation in a complete system.

3.1 Adaptive Loading Techniques

The goal for adaptive loading is to avoid the expensive and resource consuming procedure of loading the complete data set when this is not necessary. The direction is then to only partially load data and the questions to answer are: *when* we load, *how much* we load every time and of course *how* we load.

For this initial study we will for simplicity assume a columnar layout to focus purely on the loading part. This way, here when we refer to data loading, this happens at the granularity of columns (or parts of it) resulting purely in an array-based storage form underneath. Row-store formats and hybrids, they all pose slightly different challenges waiving away some of the problems a column format poses but at the same time adding new ones. The analysis below demonstrates the potential of adaptive loading and indicates potential benefits from further detailed studies.

3.1.1 When

Assuming dynamic and evolving scenarios without upfront clear knowledge of what to expect, the decision is to load dynamically and partially *during query processing*. This way, we achieve the zero cost initialization property as all loading responsibility is transferred to queries, just as it is with scripting tools.

3.1.2 How Much

Again, assuming no workload knowledge, the decision is to do the minimum possible investment at every time. This way, for each query we process, we make sure we have all necessary data in order to correctly and completely answer it. In other words, incoming queries not only trigger loading but also define how much we load based purely on their needs.

One direction here is to load complete columns at a time, having only the hot columns loaded. When a query comes, make sure that all needed columns are there and if any are missing, then go back to the flat file to load the needed part.

A second direction is to only partially load columns in order to reduce the loading overhead even more. This can be thought of as *pushing selections down into the loading phase*. Loading only qualifying data ensures a minimum possible per query overhead and a minimum possible storage footprint. This is ideal for exploratory scenarios where the user "walks" through the data space, periodically zooming in and out of specific data areas. However, an extra cost and complexity is involved in needing to maintain a table of contents so that we know what portions of a column are loaded. Naive strategies might have to go back to the flat file too often.

3.1.3 How

We can identify two topics here. First, how do we fit these techniques in the software stack of a DBMS. Second, how do we actually access the flat files in an efficient way.

We design new adaptive loading operators which are plugged into query plans and are responsible to bring the missing data. These operators mimic the pure loading procedure of a DBMS. The difference is that they can *selectively* bring data from a file. Such an operator can load, on-the-fly, any combination of columns from a flat file. We also experimented with operators that load only one column at a time. This is simpler to design but it turns out to be much more expensive due to the need to touch the flat file multiple times within a single query plan. For each column, we also keep additional metadata information that indicates whether this column is loaded or not, and if it is loaded then we maintain the information of which parts are already loaded and where and how they are stored. A tree structure that organizes the data parts of each column based on values is sufficient, e.g., an AVL-tree or a B-tree.

The general idea is that after all optimization of the original query plan is finished, a new optimizer module/rule takes over to rewrite the optimized plan into a query plan that properly contains the new loading operators. For each column that is marked as not fully loaded we potentially need to act. For each table referenced in the plan, the optimizer will add one adaptive load operator to bring in one go all missing columns or parts of them based purely on which columns or parts we miss for the current query. This way, the system can handle in the same query plan both kinds of tables (already loaded or not), as well as combinations of columns and tables which have different portions loaded at any given time.

In our current implementation and query plans, these operators are plugged in as close as possible to the operators that need the relevant data. Although this is a very reasonable first approach, one can think of optimization scenarios where the loading operators are plugged in dynamically while the query is running, when the system realizes that it has the resources to do some of the extra I/O and processing necessary. In addition, here, for simplicity we assume that the original DBMS optimizer rules are optimal for the new kernel as well. Naturally, this may not be true (at least not always) opening a research line that introduces specific optimizer rules and plans for such adaptive systems. Taking into account the expected costs and delays of adaptive loading operators, with respect to the actual work that they need to do for the current query, might lead to different decisions early in the optimizer pipeline.

3.2 Experimental Results

Here we provide early results of our implementation in an open-source column-store, MonetDB. We test our implementation against the normal MonetDB system and against the MySQL CSV engine. The latter is a recent engine option of MySQL which allows to directly query flat CSV files. It provides the flexibility of querying a flat file with SQL but it does not provide the DBMS benefits as it resembles a Unix-tool like Awk in that it needs to read the data again and again for every new query, i.e., it does not load the data in any way, optimize the layout, etc.

We implemented new adaptive loading operators and query plans in MonetDB. These operators can be plugged in query plans to provide on-the-fly loading of only the necessary data. From a high level point of view the operators work as follows. The flat file is split into multiple portions horizontally. Then, multiple threads take over to tokenize each portion. Tokenization is done in two steps for each file portion. First, we identify where each row begins. Then, within each row we find out where the relevant columns are. While

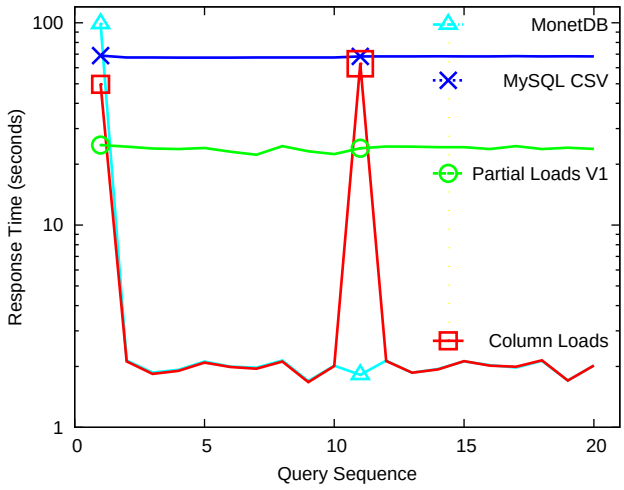


Figure 3: Alternative Loading Operators

tokenizing a given row to identify the needed columns, once all required columns are found the tokenization for this row can stop., i.e., there is no need to tokenize any columns not needed for this query. In addition, if the *where* clause is pushed into the loading operator, then once a needed attribute is tokenized, it is parsed and the relevant predicate, if any, is applied on-the-fly. This allows us to abandon the tokenization of a row as soon as a predicate fails to qualify for this row. After all tokenization is done, then multiple threads take over to parse the values and put the proper values in the proper columns and in the proper order. The process continues until all horizontal portions are read.

Figure 3 shows results on a 10^8 tuples table. It contains 4 attributes of unique integers and we use Q2 queries as follows.

```
   select sum(a1),avg(a2)
Q2 from R
   where a1>v1 and a1<v2 and a2>v3 and a2<v4
```

Each query uses two attributes and is 10% selective. Here we first run 10 random queries that use the first two attributes of the file and then we run another 10 that use the last two attributes.

The MonetDB curve represents the normal database behavior where everything is fully loaded up-front. Here, the complete loading cost is attached to the very first query, representing a significant delay. Every query after that is fast as it exploits the already loaded data (no indices are created here). The CSV engine of MySQL demonstrates a stable performance. Having to read and analyze the complete flat file for every query brings a constant response time throughout the query sequence. The Column Loads curve represents our new query plans in MonetDB that on-the-fly load the needed columns missing from the current storage. The cost of the first query is roughly half the cost of the normal database query representing a significant gain. The next 9 queries enjoy very good performance similar to that of plain MonetDB. These queries also need the same attributes as the first one, so all needed data is already there and no extra actions are needed, leading to similar performance as that of MonetDB. Query 11 though needs a completely different set of data and thus it on-the-fly needs to load the missing columns. However, again this is much faster than a complete load and all queries after that can exploit this to gain good performance. Overall, both the MonetDB curve and the Column Loads curve spent similar amount of time in loading. However, the Column Loads curve spends this cost only when it is needed, allowing for faster initialization of data exploration. The cost is spent only when and if necessary, i.e., if the workload never demanded some of the columns of the file, those would never be loaded.

Finally, the Partial Loads curve represents query plans with loading operators that can perform filtering on-the-fly. Only the values that qualify the where clause are loaded, allowing the loading operators to shed extra cost by minimizing the actions on non-qualifying data. Complete tuples in the flat file can be ignored as soon as one of the predicates in the where clause fails for a given tuple which allows the adaptive operator to immediately stop any further parsing and loading actions in this row. This kind of partial adaptive loading has the side-effect of creating intermediate results that are identical to what a selection operator over the complete column would create. This way, the query plan can then continue from this point on, avoiding all the where clause operators.

For this specific experiment, Partial Loads throws away the data immediately after every query. Thus, this represents the potential benefits of the most lightweight direction of never "really loading" anything, never paying the I/O cost to write the data back to disk and always reading just enough from the file by exploiting early tuple elimination while parsing. However, given that we have to go back to the file for every query, as MySQL does, performance remains quite stable with no room for improvement. Multiple variations of the above are possible, shaping up an optimization problem. In the next section, we present a variation where Partial Loads does not throw data away and future queries can actually reuse them if they overlap.

4. DYNAMIC FLAT FILE ADAPTATION

The previous section showed some very promising results. Loading can be done incrementally and it can be added in the software stack of a modern DBMS. Given though the inherent costs of reading and analyzing flat files, these results indicate that a significant cost is involved every time we need to read from a file due to missing data. Here, we study this problem and sketch a solution that goes even more beyond the traditional loading schemes.

4.1 Split Files On-the-fly

4.1.1 I/O Overhead

Going back over and over to the full file has the inherit cost of requiring to bring the complete file from disk, tokenize and parse a part of it over and over again. When the complete file needs to be loaded, it is justified to pay this cost. In our case, though, we are incrementally loading the data and we have to go back to the file at multiple occasions; flat files are organized in rows but the queries ask for specific columns. Thus, this is exactly the same problem as in the typical row-/column-store environment where a column-store has the advantage of selectively reading only the necessary columns for a query, while a row-store needs to read everything no matter what the query needs are.

4.1.2 Tokenization Overhead

A second problem with a monolithic flat file is that every time we need to load a different attribute or part of it, we need to tokenize *all* attributes that appear before our target in the file. For example, if we want to locate the 5th attribute in each row of the file, we need to tokenize the previous 4 so that we know when we reach the 5th one, even though these attributes are not relevant for the current query. If for a future query we need to load part of attribute 6, then again we need to tokenize all previous 5. Maintaining information on where each attribute starts is prohibitive as this is different for each row; assuming fixed length attributes is unrealistic in general. Even if we had such knowledge, we still have the inherit cost of needing to read the complete raw file from disk even if only a few of the values are needed.

4.1.3 Splitting Flat Files

The direction to follow here is to try to amortize the loading cost over the sequence of queries, making sure that we do not perform the same actions over and over again. To achieve this, we have to eliminate the need to repeatedly read the complete file. At the same time, we want to avoid tokenizing the same part multiple times. Both of these goals can be achieved if we incrementally and adaptively *split* the file during the loading phase such as future loading steps can locate the needed data much easier.

For example, say we need to load column A. With the techniques of the previous section, we have to go through every row of the file and tokenize it until we locate column A. Then, we parse and load this value and ignore the rest of the row. Thus, for every row of the file, we have paid the cost of tokenizing every attribute before A, but we have not tokenized any attribute after A.

4.1.4 Dynamic File Partitioning

To exploit our current efforts in future queries, one direction is to on-the-fly create a few supporting structures as a side-effect of loading. For example, here we can create one new flat file for each attribute we tokenized and one flat file for all attributes we did not tokenize. This way, when in future queries we need to load any attribute which is already tokenized, we can simply read the respective input file containing only the needed values. The befit is twofold. First, we completely avoid the overhead of reading the rest of the input data. Second, loading the column becomes much simpler since the file contains only the needed column and thus tokenization involves only separating the values by recognizing the end of each row. Even if we need an attribute from the non-tokenized ones, again we gain by reading and analyzing only part of the flat file and thus progressively making the loading cost less and less as we gain more knowledge.

4.1.5 Learning

Conceptually this can be thought of as "file cracking", i.e., similar to database cracking [12] that dynamically reorganizes columns inside a DBMS based on queries, file cracking dynamically reorganizes flat files to fit the workload needs and ease future requests. The concept of cracking is to learn as much as possible when touching data and with lightweight actions reflect this knowledge. The same mentality can be applied here with several research directions to study towards on-the-fly maintaining a more sophisticated table of contents over the flat files. Every time we touch a file, we learn a bit more about its structure, e.g., the physical position of certain rows and attributes. Solutions that provide a more active reorganization of the flat files are also possible in order to introduce even more knowledge. Identifying and exploiting this knowledge in the future can bring significant benefits.

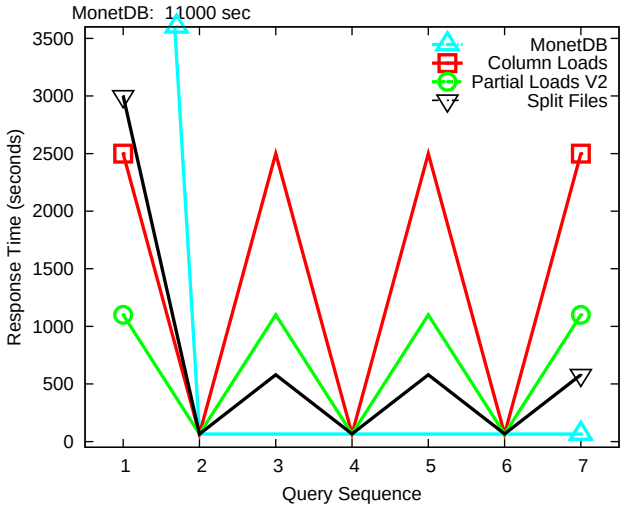


Figure 4: Adaptive loading with file reorganization

4.2 Experimental Results

We implemented the split file functionality in our MonetDB implementation of adaptive loading. During tokenization, the already seen columns which do not qualify for the current query are not ignored as before. Instead, pointers to the values of each column are collected into arrays and once all tokenization is finished, they are written in one go in one separate file per column. The non tokenized columns are written in a single separate file. The system then has to update its table of contents regarding where subsequent queries can find each column.

Figure 4 shows the results. We use a 12 column table of 1 billion tuples and Q2 queries. Every 2 queries we use 2 different attributes of the table until all attributes have been used, running 12 queries in total. In order to emphasize the best and the worst case for each technique, the second query in each run is simply a rerun of the first, i.e., we run 6 different queries, and each query runs twice. The y-axis in Figure 4 is trimmed at $3.5 * 10^3$ seconds while the first query of MonetDB reaches $11 * 10^3$ seconds. In addition, for better presentation the x-axis shows the first 7 queries as the rest of the sequence maintains the same performance patterns.

The performance of MonetDB and Column Loads is similar to our previous experiment; MonetDB loads the data completely with the first query, while Column Loads loads incrementally whenever missing columns are needed. With Column Loads we load only the necessary columns every time essentially amortizing the loading cost across the workload. Query 1 loads fully 2 of the columns. Then Query 2, needs the same columns and thus it can provide a performance similar to MonetDB as no extra actions are needed; not all data is loaded but all necessary data is there. The Partial Loads V2 manages to produce smaller peaks by se-

lectively loading only the necessary values. Unlike the experiment in Section 3, here Partial Loads maintains the data between queries providing incremental loading.

The effect of the Split Files technique is that it can produce even smaller peaks. In this case, in order to emphasize the best and worst behavior, the very first query asks for the two attributes that appear last in the flat file. This way, it is the first query that does all the heavy work of splitting the *complete* file and thus demonstrates the worst first query case. Even so, the start-up cost is roughly 4 times smaller than that of MonetDB. In the general case, the split file efforts will be amortized among many queries and thus the performance of the first query will be even more attractive. If we look at what happens after the first query, then Split Files achieves an ultimate performance similar to that of MonetDB and thus providing all performance a modern DB can promise. In addition, compared to Partial Loads and Column Loads it reduces the cost of dynamically going back to the flat file, i.e., it is 2 times faster than Partial Loads and 5 times faster than Column Loads. The gain comes purely by the need for less I/O and parsing effort. Every time something is missing, Column Loads and Partial Loads have to go back to the complete raw file, while Split Files can just read the individual files containing the minimal subset needed for the query. This time, this represents the best possible behavior given that the first query has already done all the file splitting.

4.2.1 Summary and Research Questions

The analysis above demonstrates a clear potential. It also raises several research questions, e.g., how much of the flat file should we bring inside the DBMS? By bringing more data than a query needs, we penalize single queries with actions that might never prove useful. By bringing exactly what a query needs, we have higher chances of needing to go again back to the flat file, penalizing queries that represent a shift of the workload. Splitting the file dynamically helps but it potentially doubles the needed storage budget and loses (or makes more complex) the ability to directly edit the flat files via any text editor and immediately fire an SQL query. Quantifying and modeling the various actions is also needed to take educated actions.

5. RESEARCH LANDSCAPE

In this paper, we set a challenging goal towards fully autonomous systems that consider flat files as an integral part of their structure. The goal is to achieve a zero initialization overhead; just point to the data and start firing SQL queries, progressively achieving similar performance with that of a modern DBMS.

The initial designs and experimentation shown in the previous section verify that this is indeed a feasible direction. We clearly show that what is considered as a given fact in modern database systems regarding complete and expensive up-front data loading can be reconsidered.

The task of a complete study of this vision expands over several core database topics and requires an in depth analysis. In this section, we sketch the most important of these topics towards our vision and we provide initial ideas and discussion for the adaptive storage module, the adaptive kernel module as well as for updates, concurrency control, robust performance, etc.

5.1 Adaptive Store

The storage layout is of critical importance. In our vision, storage is created on-the-fly as data is incrementally brought from the flat files into the system. In fact, the storage layer consists of two parts: (a) the flat data files and (b) the data that the engine creates to fit the workload, the *Adaptive Store*.

5.1.1 Continuous Adaptation

The adaptive store is an ever changing layer that continuously adapts to the workload. The key point here is that all this happens on-line when we actually have information that we can exploit towards making these decisions. The queries themselves provide this information. This way, we can continuously create the "optimal" representation for the query at hand.

In the adaptive store the notions of base data, index, materialized view and projection are blurred. The adaptive store may contain data in *any format*, i.e., row-store, column-store, as well as PAX and its variations. For example, while loading data dynamically and partially, instead of merely creating columns to store the data, the system can choose the optimal format for the current query that triggered this loading action.

Multi-format Data. In the same spirit as with the above observations, it is not necessary that all data for a given table follows the same format. On the contrary, different parts of a table may follow completely different formats. Queries dictate the format of the data parts they need and different parts of the same table may be of interest to different queries. By letting independent queries make independent decisions on how to store the missing relevant data and how to exploit existing already loaded relevant data, the system self-organizes to match the workload.

Multi-format Copies. The same part or overlapping data parts may be replicated multiple times in multiple different formats in order to service different kinds of queries.

Multi-file Data. A single data instance in the adaptive store may contain snippets from multiple raw data files, effectively providing partial denormalization as needed by the queries.

Data Padding. Data padding is a powerful tool to fully exploit modern hardware. For example, [15] shows that data padding techniques can successfully be integrated with execution strategies that are aware of the padding and through bitwise operations provide efficient database query processing. A system that can exploit proper padding in an adaptive way and combine this with the rest of the data placement and execution decisions discussed above, leads to an optimal performance, adapting to its hardware potential.

5.1.2 Strategies

There are two extreme strategies one may implement regarding how to manage the adaptive store. In the first one, we target for simplicity, i.e., for every incoming query that is not fully covered by the data which is already loaded, we blindly treat this query as if all the tuples it needs are missing. This means that we will fetch everything from the flat files, including tuples that potentially are already loaded. Then, we create a completely new view to store the newly loaded data. In the second extreme, we always try to minimize the footprint of the adaptive store, i.e., for every incoming query we identify the missing tuples and we load

only those. Then, we have to update the proper structures in the adaptive store to reflect the new information as well as combine the existing relevant tuples such as to answer the current query.

Naturally, the second approach is more complex than the first but it gives more flexibility to exploit the adaptive store and cover a bigger percentage of the active workload. The research challenges here are in designing efficient techniques to implement strategies like the one described above as well as study the field between those two extremes.

5.1.3 Life-time

Multiple interesting scenarios may be studied regarding how long we need to keep data alive. Data parts loaded via adaptive loading and stored in any format may be thrown away at any time. The only cost is that of having to reload this data part if it is needed again in the future.

For example, one approach can be that the adaptive store is purely memory resident. In this case, loaded data is never written to disk and stay alive only as long as there is enough memory. Once a more "useful" piece of data is needed, an old one is thrown away. Alternative approaches may exploit flash disks as an intermediate layer such as to minimize the effort when data needs to be reloaded.

Designing efficient strategies and algorithms on how to maintain data is of significant importance as it can affect drastically run time performance.

5.2 Adaptive Execution

Proper data placement adapted to query processing can give significant boost in performance by optimizing I/O and cache performance. However, in a system that supports multiple data layout formats, we also need multiple processing strategies to fully exploit the potential of the different layouts.

For example, a pure column-store uses a drastically different execution strategy than a pure row-store in order to benefit even more from the columnar layout. This translates to different operators and different data flows, leading to very different implementations of the DBMS kernel as well as the optimizer. Pure column-store operators, operate on one or at most two columns at a time. Their advantage is simple code, data locality and a single function call per operator. The disadvantage is that they need to materialize intermediate results which may prove fairly expensive when for example we need to operate on multiple columns of a single table. Row-store operators, on the other hand, operate in a volcano style passing one tuple at a time from one operator to the next. No materialization is needed but numerous function calls are required.

5.2.1 Adaptive Kernel

In the same philosophy as with the adaptive store envisioned above, here we argue towards an *adaptive kernel* where at any given time multiple different execution strategies are possible to better fit the workload.

The kernel may contain any kind of operator implementations that better fit the data in the adaptive store based on the query patterns. And should be able to create any kind of query plan to better exploit those operators.

5.2.2 Hybrid Operators

Other than using pure row-store or column-store operators, there is also a clear space for hybrid operators. For example, when we need to compute an aggregation over three attributes, a new operator that in one go computes the total aggregation would provide the best result, i.e., operating in a column-store like fashion but with a row-store like input.

Combined with optimal hybrid storage, hybrid operators are expected to have a significant impact on system behavior as it essentially means that we always get the optimal storage and access patterns.

5.3 Auto-tuning

The adaptive store and the adaptive kernel envisioned in the previous sections aim in always maintaining the best possible layout and execution mechanisms as the workload evolves. The adaptive loading procedures keep feeding the store and the kernel with just enough data as needed.

Several challenges arise to satisfy all these requirements. Probably the most challenging of all is the question of how the system reaches a good set-up as well how it adapts when the requirements change again. Up-front materialization of all possible storage patterns is not feasible given the numerous combinations leading to storage and maintenance problems. Similarly, up-front implementation of all possible operators/access patterns is not possible given the numerous combinations of inputs and operations to be considered. This way, a dynamic mechanism is needed to on-the-fly create not only the appropriate data layout but also the appropriate operators and plans to match the current workload.

The main challenge is to define and develop the adaptation mechanisms and steps that transform the raw data and the initially "ignorant" kernel into the best fit system for a given workload. Other than the technical challenges, for this we first need to answer the following (high level) questions:

1. What is the best execution pattern for a query set?
2. What is the best storage layout for a query set?
3. How and when we actually adapt?

5.3.1 Identifying Optimal Strategies

Questions 1 and 2 require an extensive benchmarking of all possible hybrid storage and execution patterns to understand and model the performance. Even though extensive and important research was introduced in recent years, still one cannot argue with certainty which storage and execution strategy is the best for a given scenario especially when hybrids strategies and execution are thrown in the equation. This is an open research question.

5.3.2 Adaptation Triggers

Question 3 goes deep into the new architecture characteristics and of course hide numerous more fine-grained questions. As we discussed earlier, in order to assist the adaptive nature, the initial choice is to trigger adaptation based purely on the query needs. Changes in the storage or execution layer become a side-effect of query processing. Nevertheless, several issues naturally arise regarding what can trigger and guide adaptation. One could consider multiple queries at a time, system and hardware parameters, etc.

5.3.3 Re-organization

The adaptive store and the adaptive kernel are ever evolving structures. They continuously change their shape to fit

the workload but also to adapt to restrictions, e.g., storage restrictions. Data in the adaptive store may be reformatted on-the-fly if the workload indicates towards a more fine-tuned storage layout. Similarly, the kernel may alter its execution strategies to follow such changes.

Furthermore, existing data and operators may be thrown away at any time to make room for new ones based on the workload. This may be done, either to deal with storage restrictions, to ease updates or simply to lead to a more lightweight and thus better performing system.

5.4 Updates and Concurrency Control

Updates and concurrency control is one of the hardest topics in database design. In our proposed vision these topics become slightly more complex due to the continuous data reorganization. There is though a plethora of interesting ways to resolve these issues.

Regarding updates, the issue is how to update an already loaded table R. There are two reasons why we would like to update such a table. The first one is that the flat file has changed, e.g., because the user updated it via editing with a text editor. The second one is that we might want to extend R with more data. Recall that in our design loaded tables are partial views of the actual data that belong in a table. The data materialized reflect the workload needs and it is based on design choices. For example, we have seen techniques in this paper, where in a column-store architecture we do full or partial loads. When we do full loads, this means that the very first query that requests a column A, will completely load column A, while with partial loads it will only load the portion of A needed. The second choice means that future queries needing A and not covered by the loaded portions, will need to update the existing structures. The second scenario brings concurrency control issues as well. Multiple queries might be asking for the same column at the same time, meaning that these queries have to touch and update the same loaded table with data brought from the flat file.

One easy solution to all the update and concurrency control scenarios is to treat each request independently and each table as a completely auxiliary data structure that we are not afraid to lose. For example, each incoming query that is not fully covered with whatever is already loaded in the database will be treated as if it is the first query for the data needed. This way, it will create its own table portions but without incurring any conflicts on the way. Similarly, every time a flat file is updated, we can simply drop all relevant tables that have been created with data from this file.

The advantage of the above approach is simplicity. The research challenge here is to investigate differential methods and proper locking strategies for updates such as to avoid extensive data replication, share as much work as possible between concurrent queries and to avoid extensive trips back to the flat files.

5.5 Robust Performance

Any adaptive method may suffer from a non robust performance. For example, the worst case scenario in a system with adaptive loading is one where the data parts loaded are never used. Say, we have a memory limit of X bytes. Queries arrive and start creating more and more tables and feed them with data from the flat files. Say that no query is covered by any existing table and thus all queries have to go back to the flat files. Then, we reach the storage limit and we start dropping tables. Thus, all the effort of incremental loading is wasted.

Another similar scenario is the case of incrementally loading a column with partial loads where each query brings from the flat file only the portions that it needs. Individually each query is much faster than one with full loads as we saw in our experiments. However, in the worst case scenario we will have to go back to the flat files as many times as the tuples in a column. Say for a column of N tuples we receive N queries one after the other where each query fetches only one of the N tuples. In this case, doing a full load with the first query is a much better approach as it avoids the extensive I/O.

Thus, the challenge, for providing a robust performance relates to a continuous process to monitor the system performance and the workload trends such as we can continuously adjust critical decisions that may significantly affect performance, i.e., how much data to fetch from the flat files with every query, whether to replicate tables or use a single instance of each one, whether split the flat files or not, etc.

5.6 Schema Detection

Another important research topic is automatic schema discovery. When the user links a collection of flat files to the database, a schema should be defined. Ideally, this should be done without any input from the user. Following a simple strategy, each flat file can be mapped to a table. In the tokenization phase, the columns of a given row are identified, each column becomes an attribute of the table and we examine each attribute to figure out its proper data type. This task is performed only once during the first query execution.

Nevertheless, all schemas are not equally good; many times integrity constraints and functional dependencies are important. Advance techniques such as database normalization, data de-duplication and data cleaning can be considered and we cannot expect that the structure of the flat files reflects a good schema for the query workload indented.

6. RELATED WORK

There have been several important milestones in the research literature that brought us to this research line. In this section, we briefly discuss related work that inspired the vision of this paper. There has been related work in multiple aspects of database research, mainly related to external tables functionality, self-organization and hybrid storage layouts.

6.1 External Tables

Flat files were always considered "outside" the DBMS engine. The general guideline is that a DBMS has to load the data completely before it can do anything fancy with it. Recently, there have been a number of efforts both in industry and in academia to partially attack this problem.

FlatSQL [16] claims the usefulness of using SQL to express queries over flat files. The main motivation is that one can use SQL, i.e., a declarative language as opposed to a scripting one to interact with the system. Under the covers, FlatSQL translates SQL to Awk scripts so it can actually query the flat files. This work was mainly motivated by scientific applications and it shows that the motivation for using a DBMS is not only raw performance but also several of the features that make DBMSs friendly to the user.

In [22], the SciDB project presents a collection of essential features needed for modern scientific databases with huge data loads. The ability to query raw data by minimizing the overhead of loading was included as one of these features that modern DBMSs need to support so they can be useful for scientific data analysis.

More recently, several open source and commercial database systems include the functionality of external tables. The idea is that the system can read directly from flat files triggered by an SQL query. Nevertheless, current designs do not support any advanced DBMS functionality. In other words, they provide the same benefits as with FlatSQL but without the need to call Awk. In terms of performance though, this cannot match a normal DBMS as it needs to continuously parse the data.

The vision provided in this paper goes multiple steps further. By allowing not only to access flat files via SQL but also to selectively and dynamically load part of the data and store it in what essentially is an index-like format, we can combine both the ease of using SQL and the superior performance of exploiting a traditional database engine.

6.2 Self-organization

Self-organization has been studied in multiple problems of database research. The main goal of any technique in this front is to reduce or eliminate the need to tune the system which in turn significantly reduces both the user input required and the time needed to reach a highly tuned system. Typically, this means an effort to automatically select and create indices, materialized views, etc.

Auto-tuning Tools. For example, there has been a significant line of work in the area of auto-tuning tools, e.g., [1, 3, 4, 20, 23, 25]. These tools automatically select the proper indices given a representative workload. They can even take into account the available storage budget that we can devote in the auxiliary structures, the cost of updates, etc. Nowadays, these are invaluable tools when setting up a new system. This is typically an off-line approach, i.e., it requires a priori workload knowledge and enough idle time to analyze the representative workload. More recently, there have been efforts in adapting these tools towards a more on-line approach [4, 20]. In this case, the system can start with zero indices, then monitor incoming queries and performance and eventually come up with a set of candidate indices that match the running workload.

Adaptive Indexing. A second line of work is adaptive indexing, i.e., database cracking and adaptive merging [7, 8, 9, 14, 12, 13]. With such adaptive indexing techniques, index selection and index creation happens as a side-effect of query processing. The user does not have to initiate any tuning or provide a representative workload. At any given point in time, an adaptive index is only partially optimized and partially materialized such as to fit the current workload and storage budget. As queries arrive, the index representation adapts at the physical level to fit the workload. For example, the basic cracking techniques can be seen as an incremental quick sort whereas the basic adaptive merging techniques can be seen as an incremental external sort. For example, in the MonetDB implementation of database cracking and adaptive merging in a column-store, each operator (a selection, a join, etc.) physically reorganizes the arrays that represent its input columns using the query predicates as an advice of how the data should be stored.

Blink. Another recent research path is the Blink project from IBM [15, 19, 17]. Recognizing that index selection, exploitation and tuning is a major hurdle, Blink completely removes these steps. The motivation is that the system should be usable with minimum tuning; there are no indices and there is no optimizer. Blink combines several novel techniques to deliver high performance out of the box. Some of the core technologies include denormalization, compression via frequency partitioning and a run-time kernel that can operate directly on compressed data. Frequency partitioning allows to create multiple partitions of the data, collecting similar values from each attribute in the same partition and compressing them with fixed length codes per partition. This way, Blink achieves both high compression and high performance, since at run time the kernel can exploit vectorized query processing. In addition, Blink packs several tuples into CPU registers and operates on blocks of tuples at a time even in compressed form. The end result is high performance with zero or minimal tuning; the user needs to simply load the data.

Our Approach. All research effort described above, relates to ours in that it tries to improve the user experience regarding the tuning effort needed. However, our vision goes a significant step further to introduce the self-organization flavor all the way to the very beginning of the user experience, i.e., *before even loading the data*. All existing efforts help only after the data is completely loaded. We expect that several of these ideas will apply to our vision as well.

6.3 Storage Layout

Trying to create a hybrid store has been the subject of several seminal papers over the years, e.g., [2, 10, 18, 11, 21, 24]. Lately, the issue has received wide recognition by the commercial world as well, with big vendors and new start-ups pushing out hybrid technologies, e.g., Oracle, Greenplum, Vertica and Vectorwise. Previous work was mainly focused on improving the way data is stored, bringing the I/O or cache benefits of a column-store design to a row-store setting but leaving the execution part as is, i.e., using N-ary processing. Other work [26] also demonstrates that significant benefits may come from combining the NSM and DSM execution strategies. Building on top of a pure column-oriented storage layout, it shows that it is beneficial to change between NSM and DSM execution strategies on-the-fly. In fact, attempts for hybrid designs began even earlier when vertical partitioning was the main tool to improve performance with respect to the I/O, e.g., [5].

More recently [6] proposes a declarative language interface to explicitly define storage patterns and data layouts. Such an interface would be of significant importance since it can simultaneously simplify and model the interaction with the lower storage level. It still requires, however, a set-up step which assumes a good grasp of the query and data workload. Thus, this is orthogonal to our vision here.

Contrary to the above literature, we sketch the vision of an adaptive store that automatically feeds itself from flat files, making sure that data fits the workload both in terms of which data parts are loaded and also in terms of which storage and access pattern is being used for each data set.

7. CONCLUSIONS

This paper sets a challenging vision; the user should just give the raw data files as input and the system should be

immediately usable. Flat files should not be considered outside the DBMS anymore. Here, we sketch this research path and provide an analysis of how a system can adaptively and dynamically load only parts of the input data, and keep feeding from flat files whenever this is necessary. Hybrid storage and execution techniques will further enable this vision that opens a research line towards systems where each incoming query is treated as a guideline of how to load, how to store and how to access data.

In addition to opening up a new line of research, we expect a significant impact by enabling wide database systems usage across multiple domains. Scientists benefit from faster and (more) complete data analysis thanks to shorter query response times. At the same time, several other application domains can benefit ranging from large corporate set-ups to everyday personal applications. The latter should not be ignored! For example, a person's music or photo collection is typically stored in a file hierarchy, manually organized. With personal data growing in massive numbers and ranging over numerous areas, e.g., music, photo, digital books, movies, agendas, maps, etc., personal data management quickly becomes a horrendous task – but a single user will never go into the trouble of putting his/her data into a DBMS due to the initialization trouble and expert knowledge required (the interface should also change from SQL to natural language but this is an orthogonal issue).

Thus, the path of truly adaptive and autonomous databases applies to a vast range of scenarios and has the potential to have a significantly positive effect on modern life.

8. REFERENCES

[1] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server. In *VLDB*, 2004.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[3] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*, 2005.

[4] N. Bruno and S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB*, 2006.

[5] D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Software Eng.*, 16(2), 1990.

[6] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[7] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *TPCTC*, 2010.

[8] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *SMDB*, 2010.

[9] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[10] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. VLDB '03.

[11] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, 2006.

[12] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[13] S. Idreos, M. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, 2007.

[14] S. Idreos, M. Kersten, and S. Manegold. Self-organizing Tuple-reconstruction in Column-stores. In *SIGMOD*, 2009.

[15] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[16] K. Lorincz, K. Redwine, and J. Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS. *Harvard*, 2003.

[17] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1), 2008.

[18] R. Ramamurthy, D. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[19] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, 2008.

[20] K. Schnaitter et al. COLT: Continuous On-Line Database Tuning. In *SIGMOD*, 2006.

[21] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *VLDB*, 2004.

[22] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *CIDR*, 2009.

[23] G. Valentin et al. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, 2000.

[24] J. Zhou and K. A. Ross. A Multi-Resolution Block Storage Model for Database Design. In *IDEAS*, 2003.

[25] D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

[26] M. Zukowski, N. Nes, and P. A. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, 2008.

The SQL-based All-Declarative FORWARD Web Application Development Framework [*] [†]

Yupeng Fu
UC San Diego
yupeng@cs.ucsd.edu

Kian Win Ong
app2you Inc.
kianwin@app2you.com

Yannis Papakonstantinou
app2you Inc. / UC San Diego
yannis@cs.ucsd.edu

Michalis Petropoulos
SUNY Buffalo / UC San Diego
mpetropo@buffalo.edu

1. INTRODUCTION

The vast majority of database-driven web applications perform, at a logical level, fundamentally simple INSERT / UPDATE / DELETE commands. In response to a user action on the browser, the web application executes a program that transitions the old state to a new state. The state is primarily persistent and often captured in a single database. Additional state, which is transient, is maintained in the session (e.g., the identity of the currently logged-in user, her shopping cart, etc.) and the pages. The programs perform a series of simple SQL queries and updates, and decide the next step using simple if-then-else conditions over the state. The changes made on the transient state, though technically not expressed in SQL, are also computationally as simple as basic SQL updates.

Despite their fundamental simplicity, creating web applications takes a disproportionate amount of time, which is expended in mundane data integration and coordination across the three layers of the application: (a) the visual layer on the browser, (b) the application logic layer on the server, and (c) the data layer in the database.

Challenge 1: Language heterogeneities. Each layer uses different and heterogeneous languages. The visual layer is coded in HTML / JavaScript; the application logic layer utilizes Java (or some other language, such as PHP); and the data layer utilizes SQL. Even for pure server-side / pure HTML-based applications, the heterogeneities cause impedance mismatch between the layers. They are resolved by mundane code that translates the SQL data into Java objects and then into HTML. When the front end issues a request, code is again needed to combine memory-residing objects of the session and the request with database data. Consequently, developers write a lot of "plumbing" code.

As a data point, in a UCSD web development class taught by the authors, the students built a web application of their choice. A class project averages 4700 lines of code and configuration, but for 1 line of SQL, there is a modest 1.5 lines of Java used for business logic, and 61 lines of Java used for plumbing.[1] That is a lot of plumbing code to write for the computationally simple functionality that is typically required by the majority of web applications!

Challenge 2: Updating Ajax pages with event-driven imperative code. Since 2005, Ajax led to a new generation of web applications characterized by user experience commensurate to desktop applications. The heavy usage of JavaScript code for browser-side computation and browser / server communication leads to superior user experience over pure server-side applications, comprising

- performance gains through partial updates of the page
- more responsive user interfaces through asynchronous requests, and
- rich functionality through various JavaScript component libraries, such as maps, calendars and tabbed dialogs.

For example, consider the web application page of Figure 1, where each user submits proposal reviews, reads the reviews provided by the other reviewers and also views a bar chart of the average grades for each proposal. In a pure server-side model, submitting a review for a single proposal will cause the entire page to be recomputed. Indeed, queries will be issued for the reviews and average grades of all proposals, not only for the reviewed proposal. Furthermore, the browser will block synchronously and blank out while

[*]Supported by NSF awards IIS 1018961, IIS 0917379 and a Google gift.

[†]The license grant at the bottom of the first column does not confer by implication, estoppel or otherwise any license or rights under any patents of authors or The Regents of the University of California.

[1]We count as business logic Java lines that decide the control flow of the application. We count as plumbing the Java lines responsible for binding SQL to Java and Javascript and vice versa, plus all code responsible for managing language and data structure heterogeneities.
A contributing factor to the huge plumbing to business logic ratio is the best practice usage of MVC frameworks such as Struts, which is encouraged by the class and promotes code modularity and reuse by separating the data model, business logic and user interface, but in turn creates more layers to copy data between.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

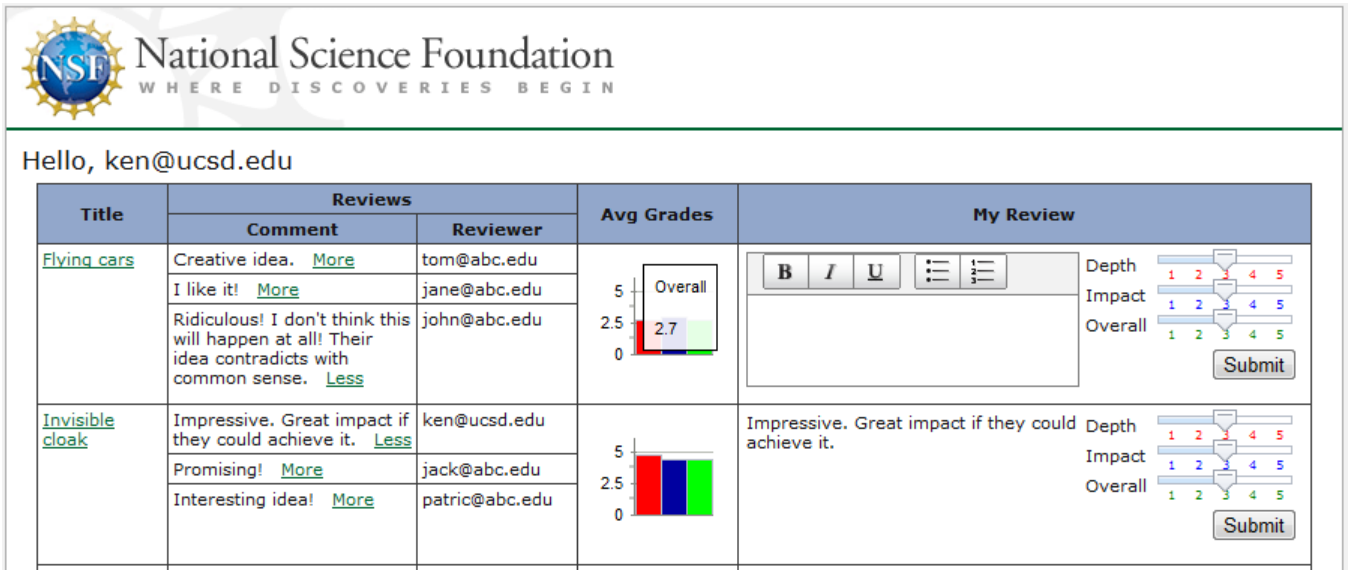


Figure 1: The review page of the running example

it waits for the new page from the server. Finally, various aspects of the browser state, including the data of non-submitted form elements, cursor positions, scroll bar positions and the state of JavaScript components will be lost and re-drawn, thus disorienting the user.

For an Ajax page, however, a developer will typically optimize his code to realize the benefits of Ajax and solve the pure server-side problems listed in the previous paragraph. The same user action (e.g., the review submission) causes the browser to run an *event handling* JavaScript function collecting data from the page's components relevant to the action (i.e. the proposal id, the review text and grades), and send an asynchronous *Xml Http Request (XHR)* with a *response handler* callback function specified. On the server, the developer implements queries that only compute the changed data (i.e. the newly inserted review and the average grade of the corresponding proposal), to take advantage of more efficient queries, as well as to conserve memory and bandwidth. While the asynchronous request is being processed, the browser keeps showing the old page instead of blanking out, and even allows additional user actions and consequent requests to be issued. When the browser receives the response, the response handler uses it to partially update the page's state. The partial update retains non-submitted forms, scroll bar positions, etc, therefore allowing the user to retain his visual anchors on the page.

The page state primarily consists of the browser DOM, which captures the state of HTML form components such as text boxes and radio buttons, and the state of the JavaScript variables, which are often parts of third-party JavaScript components. Therefore, the developer implements the response handler by writing imperative code that navigates the DOM and JavaScript components, and invokes JavaScript methods causing the DOM and components to incrementally render to the browser.

The Ajax optimizations demand a serious amount of additional development effort. For one, realizing the benefits of partial update requires the developer to program custom logic for each action that partially updates the page, which was not the case in pure server-side programming. In particular, in a pure server-side implementation, the developer needs to write code for the effect of each individual action on the database, but writes only one piece of code that generates the page according to the database state and session state. For example, suppose that the page of Figure 1 also provides a "Delete" (Review) button. The developer will have to write two pieces of code that modify the database when "Submit" is clicked and when "Delete" is clicked, respectively. The former issues an INSERT or UPDATE command while the latter issues a DELETE. Both of them share the same piece of code that generates the page showing the list of proposals, their reviews and the average grades. This piece of code is independent of what user action caused the re-generation of the page.

In contrast, in an Ajax application, each user action needs its own code to partially update the page. This piece of code consists of server-side code that retrieves a subset of the data needed for the page update, and browser-side JavaScript code that receives the data and rerenders a sub-region of the page. In the running example of Figure 1, a different piece of code would be needed for the "Submit" and the "Delete" (Review) buttons.

Such event-driven programming (which also occurs in Flash etc.) is well-known to be both error-prone and laborious [12], since it requires the developer to correctly assess the data flow dependencies on the page, and write code that correctly transitions the application from one consistent state to another. Moreover, in a time-sensitive collaborative application (similar to Google Spreadsheet) where many users work concurrently, these dependencies may extend beyond the page of the reviewer who submitted the review, and into the pages of other reviewers who are viewing the same proposal on their browsers. For example, if the developer had issued a query for `Average Grade`, but not `Reviews`, the page will display inconsistent data if another review had been concurrently inserted into the database.

Further compounding the custom logic required for each action is the amount of imperative code that needs to be implemented on the browser. Whereas the developer of a pure server-side application needs to understand only HTML, the developer of an Ajax application needs to integrate JavaScript as yet another language, understand the DOM in order to update the displayed HTML, and write code that refreshes the JavaScript components' state based on the nature of each partial update. Since there is no standardization between the component interfaces between different third-party libraries, the developer is left to manually integrate across these disparate component interfaces.

Challenge 3: Distributed computations over both browser-side and server-side state. In response to a user action, the browser sends a HTTP request to the server and activates a program, which needs access to relevant data on the page in order to perform computations that involve such page data and the database. Writing code that involves both page data and server-side data was already mundane and time consuming in pure server-side applications and became even more so in Ajax and Flash.

In a pure server-side application, the browser is essentially stateless since all state is lost when the new page is loaded. Using HTML markup such as `<input type="text" name="review" />`, the developer declaratively specifies the value collected by the textbox will appear as the parameter `review` in the HTTP request. The good news about pure server-side programming is that when a user action causes an HTTP request, the browser is responsible for navigating the DOM, and marshalling the request parameters according to the HTML specification. The bad news is that, on the server-side, the application first unmarshalls the request parameters by using Java (or PHP, etc.), and then typically issues SQL queries where the request parameters become parameters of an SQL statement. Overall, lines of Java code are expended in such trivial "extract parameter from the request, plug it in the query" tasks. With Ajax it gets much worse, as discussed next, since the marshalling of request parameters is no longer automatic.

In particular, in an Ajax application the browser maintains state across HTTP requests. Consequently, the state of the web application becomes distributed between the browser and the server. The developer is responsible for defining a custom marshalling format for the XHR request, typically in XML or JSON, and for writing imperative code to navigate over the DOM and marshall the relevant page data that must be sent to the server along with each HTTP request. The usage of JavaScript components (calendars, etc.) on the page further complicates the issue by requiring custom code that converts between the state of the component and the marshalling format. On the server-side, the developer writes custom code to unmarshall the request parameters, and then continues along the usual path, plugging such parameters into SQL statements.

A select few web application frameworks, such as Echo2 [7] and Microsoft's ASP.NET [2], mitigate the issue of distributed application state by automatically maintaining a mirror of the browser state on the server. Such synchronization can occur efficiently, using reduced bandwidth, by having the server send to the browser only the difference between the previous page state and the new page state. Yet, mirroring is only a half-solution, since the mirror made available on the server contains the exact and full state of the browser, despite the fact that each request cares about a different subset of page data. For example, the submission of a review for the first proposal of Figure 1 requires the data collected by the top editor and sliders, while the submission of a review for the second proposal requires the data of the bottom editor and sliders. Furthermore, the mirrored browser state include visual styling details, which do not matter when form data are collected, yet they trouble collection. For example, to read the values of the three sliders Depth, Impact and Overall in Figure 1, using a mirror-based Ajax framework, the developer will need to navigate through its ancestor `<div>` and `<table>` HTML elements that are used purely to specify visual layout. The net effect of these issues is that the developer's code includes many mundane lines that navigate around the extraneous information in order to obtain the relevant data for the request.

1.1 FORWARD's Declarative Solution

The data management field has recently applied with success and great promise declarative data-centric techniques in network management and games. In a similar fashion, FORWARD adopts an SQL-based, declarative approach to Ajax web application implementations, going beyond prior approaches such as Strudel [8] and WebML [3] that focused on pure server-side data publishing applications. In particular, FORWARD removes the great amount of Java and JavaScript code, which is written to address the challenges above, and replaces them with the use of SQL-based languages to facilitate integration and enable automatic optimization. The objective is to "make easy things easy and difficult things possible".[2]

FORWARD is a rapid web application development framework. The web application's pages are declaratively specified using *page configurations*. The programs that run when a request is issued are also declaratively specified, using *program configurations*. Both page configurations and program configurations are based on a minimally enhanced version of SQL, called *SQL++*, which provides access to the *unified application state* virtual database that, besides the persistent database of the application, includes transient memory-based data (notably session data and the page's data). The application runs in a continuous program / page cycle: An HTTP request triggers the FORWARD interpreter to execute a program specification. The program reads and writes the application's unified state and possibly invokes services that have side effects beyond the unified application state. (e.g., send an email). The program typically ends by identifying which page will be displayed next. FORWARD's interpreter creates a new page according to the respective page configuration. The page specification also specifies programs that are invoked upon specified user events. The invocation of a program restarts the program / page cycle.

The key contributions of FORWARD are:

- The use of SQL++ allows unified access to browser data and server-side data, including both the database and application server main memory (e.g., session). In conventional web application programming, such access would require Java and Javascript. FORWARD

[2]This objective is not followed by today's web application development frameworks. Paradoxically, powerful Turing-complete low-level imperative languages (such as Java and PHP) accomplish tasks that can be easily accomplished by appropriate SQL-based declarative languages.

eliminates Java and JavaScript from the majority of web applications, therefore resolving Challenges 1 and 3.

- The *page configurations* are essentially rendered views that visualize dynamic data generated by SQL++ and are automatically kept up-to-date by FORWARD. The specifications enable Ajax pages that feature arbitrary HTML and (pre-packaged) Ajax / JavaScript visual units (e.g. maps, calendars, tabbed windows), simply by tagging the data with tags such as `<map>`, `calendar`, etc. The AJAX pages are automatically and efficiently updated by the FORWARD interpreter by appropriately extended use of incremental view maintenance. The FORWARD developer need not worry about coordinating data from the server to the page's Ajax components, which resolves a "Challenge 2" problem.

- The business logic layer is specified by *program configurations*, where business logic decisions and the transfer of data between the database and services are expressed in SQL++, or, even simpler, in mappings, which are translated to SQL++. The program configurations have easy unified access to both the browser data and the database, because FORWARD guarantees that the browser's page data are correctly reflected into the unified application state's page data before they are used by the programs. The automatic reflection resolves Challenge 3.

- The page and program configurations have unified access to the persisted database, the page and the session via a single language (SQL++), therefore resolving Challenge 1. JavaScript needs to be written only if one needs to create a custom visual unit. Java needs to be written only for computations not easily expressible in SQL. For example, one can build the entire Microsoft CMT in the SQL++ based page and program configurations, except for the reviewer/paper matching step, which requires a Java-coded stable matching algorithm to assign papers to reviewers according to their bids.

We argue for the effectiveness of the approach by providing a demo that shows an order of magnitude lines-of-code reduction. Furthermore, Section 5 describes ongoing and future work that will further push the advantages of declarative computation by automating optimizations.

The FORWARD project was recently licensed by app2you Inc, which is a provider of a Do-It-Yourself (DIY) platform [11]. App2you's DIY platform is being refactored so that it produces FORWARD code as the user builds pages with the DIY tool. In the meantime, app2you Inc has recently deployed earlier versions of FORWARD in three human-centric business process management applications and is currently in the process of deploying the presently-described FORWARD version in (a) an analytics-oriented large-scale business intelligence application in the pharmaceutical area, and (b) a business process management application involving hundreds of pages collecting data via smart forms.

2. RUNNING EXAMPLE

The running example, which is also the accompanying CIDR demo, is a proposal reviewing Ajax application. The paper focuses on its `review` page (see Figure 1), where the reviewers submit reviews that consist of a comment collected using a web-based text editor, and depth, impact and overall grades collected using slider Javascript components. The editor appears either in edit mode (e.g. first proposal of Figure 1) or in display mode (e.g. second proposal). The page reports the titles of the proposals as hyperlinks that lead to the proposals' pdf, the full set of reviews using a Javascript component with the familar "More" and "Less" buttons, and a bar chart with the average grades.

The full version of the reviewing application is available online at `demo.forward.ucsd.edu/reviewing`, where instructions are provided on how to use it in any of the three roles it supports (namely, "applicant", "reviewer", "chair"). The application's FORWARD specification is available at `demo.forward.ucsd.edu/reviewing/code`. Online instructions on `demo.forward.ucsd.edu` also teach how to create your own FORWARD application.

The implementation of the discussed review page required 102 lines of FORWARD page specification and the implementation of review submission required 42 lines of FORWARD action specification. Building the same required 1,642 lines of HTML, Java, Javascript and SQL code in a "quick-and-dirty" Ajax application where the code is compact, often against the principles of good MVC coding, which requires separation of the control flow part of the code from the visual/interactive part of the code. In a disciplined MVC-based implementation it would take even more.[3]

3. CREATING AN APPLICATION

A developer creates an application by providing to the FORWARD interpreter source configurations, schema definitions, page configurations and program configurations.

3.1 Sources, Objects and Schemas

The source configurations dictate the type of the sources (e.g., relational database, LDAP server, spreadsheet) and how to connect to them. As a convenience for the development of cloud-based applications and databases, FORWARD implicitly provides to each application an SQL++ database source named `db`. In the running example, `db` provides the full persistent data storage of the application.

A FORWARD application's programs and pages also have access to the `request`, `window` and `session` main memory-based sources, which in the spirit of the session-scoped attributes provided by application servers, have limited lifetimes and there may be more than one instances of them at any time. For example, the `session` source lives for the duration of a session. All the pages of a browser session and all the programs initiated by http requests of the browser session have access to the same instance of the `session` source, while pages and actions of other browser sessions have their own `session` instances. Similarly the `request` source lives for the duration of processing an http request by a program (as discussed in Section 3.2) and each in-progress program has its own `request` instance. A `window` source lives for the duration of a `browser` window and becomes available to all pages and programs of such window.

[3] Note that current MVC frameworks interact very poorly with Ajax due to their literal adherence to the program-page cycle. In light of the MVC frameworks' mismatch to Ajax functionalities we did not attempt measuring how many lines of Struts or Spring would be required to build the running example.

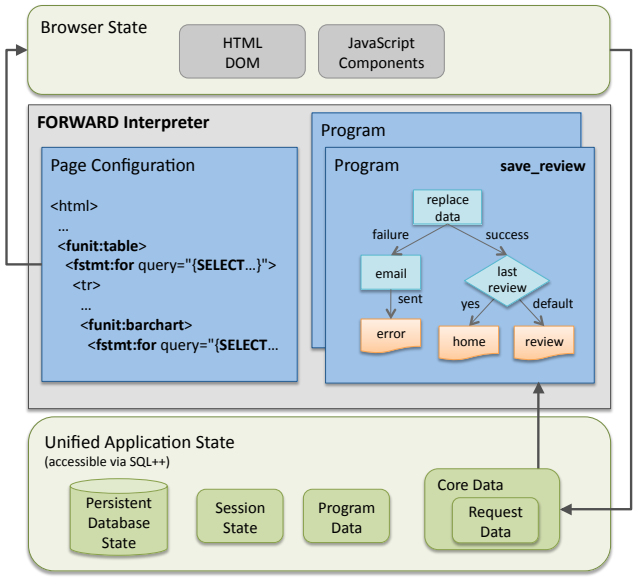


Figure 2: The FORWARD interpreter hosts applications that consist of pages and programs

Each source stores one or more objects. Each object has a *name*, *schema* and *data*. The schema may be explicitly created with the Data Definition Language (DDL) of SQL++ or may be imported from the source. For example, the schema of a relational database source is imported from its catalog tables. The DDL of SQL++ is a minimal extension of SQL's DDL.

3.2 Operation: The Program/Page Cycle

In the spirit of MVC-based frameworks such as Struts and Spring, a FORWARD application's operation is explained by program-page cycles. (In that sense FORWARD's programs correspond to Struts' actions.) An http request triggers the interpreter to run the program configuration that is associated with the request's URL. The program reads and writes the application's unified state (i.e., database, request data, session data) and possibly invokes services that have side effects beyond the application's state (e.g., sends an email). The program's run typically ends with identifying which page p will be displayed next. Conceptually, FORWARD's intepreter creates a new page according to p's page configuration, which may be thought of as a rendered SQL++ view definition, and displays it on the browser. A displayed page typically catches browser events (such as the user clicking on a button, mousing over an area, etc; or a browser-side timer leading to polling) that lead to action invocations (via http runs) therefore continuing the program-page cycle.

Notice that FORWARD enforces the full separation of the Controller functionality from the View functionality, which current MVC frameworks only encourage but do not enforce: The page configurations are literally views, unable to side effect the application state.

3.3 The Page Configuration

A page configuration is an XHTML file with added:

1. FORWARD *units*, which are specified as XML elements in the configuration and are rendered as maps, calendars, tabbed windows and other Javascript-based components. Internally FORWARD units use components from Yahoo UI, Google Visualization and other libraries and wrap them so that they can participate in FORWARD's pages without any Javascript code required from the developer.

2. SQL-based inlined *expressions* and `for` and `switch` *statements*, which are responsible for dynamic data generation.

Figure 3 provides the page configuration of the `review` page. Figure 3 excludes a few parts, which are marked by `<!-- in demo -->` and can be found on the online demo. The complete page configuration's size is 102 lines.

Lines 2-6 of the page configuration list the HTML that generates the top of the page and contains the FORWARD unit `funit:table`.[4] Notice that a unit may contain other units; e.g., the `funit:table` (line 6) contains the `funit:bar_chart` (line 19), the `funit:editor` (line 33) and the `funit:slider` (line 55). A unit may also contain XHTML, which may, in turn, contain other units.

A `fstmt:for` statement evaluates its query and for each tuple in the result (conceptually) outputs an instance of its body configuration, i.e., of the XHTML within the opening and closing `fstmt:for` tags. For example, the `fstmt:for` on line 11 outputs for each proposal one instance of the `tr` element on line 15 and its content.

The syntax and semantics of the `fstmt:for` and `fstmt:switch` statements are deliberately similar to the `forEach` and `choose` core tags of the popular JSP Standard Tag Library (JSTL) [10]. The same applies for FORWARD expressions and JSTL expressions. However, FORWARD's `fstmt:for` iterates directly over a query, whereas JSTL's `forEach` iterates over vectors generated by the Java layer, which are in turn produced by iterating over query results. Besides the obvious code reduction resulting from removing the Java middleware, we will see many more important benefits that are delivered because FORWARD analyzes the queries behind the dynamic data of the page.

FORWARD's page configurations enable nested, correlated structures on the pages. In particular, the query of a `fstmt:for` statement found within the body configuration of an enclosing `fstmt:for` may use attributes from the output of the query of the enclosing `fstmt:for`. For example, the table `reviews` has a foreign key `proposal`. For each "proposal" instance the correlated query on line 20 produces a tuple with the average grades of its reviews by using the `proposal_id` attribute of its enclosing query. Furthermore, the page configurations allow variability (e.g., the current user's review appears either in edit mode or in display mode) utilizing the `fstmt:switch` statement (line 34).

Expressions, which are enclosed in { }, can reference attributes of the query's output. Furthermore, an expression may be itself a query. In the interest of flexibility, which has been the norm in tools and languages for web page creation,[5] FORWARD coerces the types produced by the expressions to the types required by the FORWARD units or XHTML, depending on where the expression appears. Therefore, the

[4]Its syntax is identical to the standard HTML table but, unlike the HTML table, it aligns the columns of its nested tables.

[5]For example, JSP pages convert JSP expressions into strings whenever possible.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <html xmlns:funit="http://forward.ucsd.edu/units"
3        xmlns:fstmt="http://forward.ucsd.edu/statements">
4    <!-- in demo: html for css, images, headers, footers-->
5    Hello {session.user}!
6    <funit:table>
7      <th> <!-- table headers -->
8        <td>Title</td>
9        <!-- in demo: rest of the headers -->
10     </th>
11     <fstmt:for name="proposals"
12                query="{SELECT id AS proposal_id, title, pdf_link
13                        FROM db.proposals
14                        ORDER BY proposal_id}">
15       <tr>
16         <td><a href="{pdf_link}">{title}</a></td>
17         <!-- in demo: comments and reviewers -->
18         <td> <!-- bar chart showing average grades -->
19           <funit:bar_chart>
20             <fstmt:for query="{SELECT AVG(depth) AS avg_depth,
21                                       AVG(impact) AS avg_impact,
22                                       AVG(overall) AS avg_overall
23                                FROM db.reviews
24                                WHERE proposal = proposal_id}">
25
26               <bar bar_value="{avg_depth}" color="red"/>
27               <bar bar_value="{avg_impact}" color="blue"/>
28               <bar bar_value="{avg_overall} " color="green"/>
29             </fstmt:for>
30           </funit:bar_chart>
31         </td>
32         <td>
33           <funit:editor name="my_review">
34             <fstmt:switch>
35               <fstmt:case condition="{EXISTS (
36                   SELECT * FROM db.reviews
37                   WHERE reviewer = session.user
38                   AND proposal = proposal_id)}">
39                 <text>{SELECT comment FROM db.reviews
40                        WHERE reviewer = session.user
41                        AND proposal = proposal_id}</text>
42                 <state>display</state>
43               </fstmt:case>
44               <fstmt:else>
45                 <text>Type your comment here</text>
46                 <state>edit</state>
47               </fstmt:else>
48             </fstmt:switch>
49           </funit:editor>
50         </td>
51         <td> <!-- sliders -->
52           <funit:table>
53             <tr>
54               <td>
55                 <funit:slider min="1" max="5" name="depth"
56                               value="{SELECT depth FROM db.reviews
57                                       WHERE reviewer = session.user
58                                       AND proposal = proposal_id}"/>
59               </td>
60             </tr>
61             <!-- in demo: sliders for impact and overall grades →
62             <funit:button onclick="save_review">Submit</funit:button>
63 <!-- in demo: several closing tags -->
```

Figure 3: The page configuration of `review`

developer need not worry about fine discrepancies between the types used in the database (which are dictated by the business logic and are often constrained) and the types used for rendering, which are often as general as they can be. For example, the expression that feeds the `value` attribute of the `funit:slider` on line 56 is a tuple that has a single integer attribute. However, it is coerced into a float, which is the type of argument that the `funit:slider` expects.

The page as an automatically updated rendered view Conceptually, the page configuration is evaluated after every program execution. While such an explanation is simple to understand, it is only conceptual. If the page that is displayed on the browser window before and after a program's execution is the same, then FORWARD will incrementally update only the parts of it that changed, therefore achieving the user-friendliness and efficient performance of Ajax pages, where the page does not leave the user's screen (and therefore avoids the annoying blanking out while waiting for the page to reload) but rather incrementally re-renders in response to changes. [9] explains how FORWARD utilizes incremental view maintenance in order to efficiently and automatically achieve pages as incrementally rendered views.

To XQuery or not to XQuery The page configuration syntax and semantics of FORWARD closely resemble those of XQuery, as they both enable nesting, order, variability and coercions. Indeed, FORWARD allows similarity to XQuery to become even more obvious by allowing omission of the SELECT clause of the queries, which is tantamount to an automatic `SELECT *`. We have chosen SQL for three reasons:

1. The typical query input is the application's database, which is almost always relational and therefore we do not need to complicate the query language semantics with conventions on how the relational database is wrapped into XML. In that sense, the page configuration language is close to the SQL/XML relational input and XML output approach.

2. A main audience of our approach are SQL developers who feel frustrated by how hard it is to create a rendered view. There is no significant number of XQuery Web developers yet.

3. Numerous processing, optimization and consistency checking algorithms around the page configuration are amenable to cleaner reductions to respective SQL problems, without subtle complications that XQuery's semantics introduce.

Nevertheless, a limited XQuery implementation is also in the plans, as XQuery provides high power in certain cases, such as XQuery-Text.

Summary The page configuration is essentially an SQL view embedded into a visual template, consisting of HTML and funit tags. Therefore the page configuration resolves Challenge 1, since it enables the production of pages without requiring Java and javascript code in addition to SQL. Furthermore, it is implemented as an Ajax page that is automatically updated to reflect the database state, therefore resolving Challenge 2 of Ajax application programming.

3.4 Unified Application State

The FORWARD unified application state, in addition to the conventional persistent data that an application has, also includes transient application data, such as an automatically-created logical-level representation of page data, which are heavily used in web application programming.

Structurally, the unified application state consists of the set of all sources, such as the `session`, `db` and `core`, which

are described by an SQL++ schema. In order to accommodate the needs of pages, of session data and of other data typically occurring in web application programming, SQL++ is a minimal extension of SQL, whereas each schema is a tuple. An attribute of the tuple may be either a scalar type or a table, whose tuples have attributes that may recursively contain nested tables. Notice that standard SQL corresponds to the case where a schema is simply a tuple of tables (think of SQL's table names as being the attribute names of the top level tuple) and each table's tuples may only have scalars (as opposed to nested tables). In order to allow variability in the spirit of XQuery and OQL, SQL++ also supports an OQL-like union construct.

An example of a schema that uses the extra features of SQL++ is the session, which in the running example is simply a tuple containing only the standard scalar attributes `session_id`, `user` and `role` that are set by FORWARD's session management and authentication/authorization utilities (not shown). The expression {`session.user`} on line 5 is a SQL++ query accessing the attribute `user` of the tuple of the `session` schema. A key integration contribution of FORWARD, which attacks Challenges 1 and 3, is the ability of SQL++ to combine persistent data with transient data using just a single SQL++ query. For example, consider the queries on lines 36 and 39 that combine the `session.user` with the table `reviews` of the persistent schema `db` to produce the reviews of the currently logged-in user in just three lines of SQL. More integration contributions are made by the core schema and data, discussed next.

The *core* schema captures in an SQL++ schema the subset of page data that have been named by the developer, using the special tag `name`, in the page configuration. Therefore the core enables the developer to resolve part of Challenge 3, by enabling him to create a data structure encompassing the data of interest to the programs as opposed to, say, visual details. FORWARD infers the core schema by inspection of the page configuration. In the running example, the core schema, which happens to fall within standard SQL, is a table of the proposals that appear on the screen along with their reviews and grades. In particular, it is a table named `proposals` (due to the `fstmt:for` on line 11) with a string attribute named `my_review` (due to line 33) and float attributes `depth`, `impact` and `overall` due to the sliders (the last two not shown in Figure 3). The table `proposals` also has the key attribute `proposal_id` so that one can associate the data collected by the multiple instances of the editor and the sliders with proposals. Mechanically, FORWARD infers this attribute to be the key of the query that feeds `proposals` using a straightforward key inference algorithm. Notice that such inference relies on the underlying `db.proposals` table having a known key, which is an unavoidable assumption of the running example, no matter what technologies one uses to implement it.

Notice that the algorithm that infers the core schema utilizes statements and queries of the page configuration, i.e. the page's logical aspects, while it mostly ignores the unit structure and XHTML (the visual aspects). The only unit aspect that matters is the types of data collected by the user, i.e., string from the `funit:editor` and floats from the `funit:sliders`. This is a key advantage over page mirror-based frameworks, such as Microsoft's ASP.NET, that offer to the developer a server-side mirror of the page data, so that the developer does not have to code in Javascript. Unfortunately, the structure of the mirror follows the (typically very busy) visual structure of the page, as opposed to the data structure that best fits the database (a typical "Challenge 3" problem). In the running example, had we followed ASP.NET's approach, instead of having three attributes names named `depth`, `impact` and `overall`, corresponding to the three sliders, we would have a hard-to-use nested table whose first tuple would be implicitly assumed to contain the `depth`, its second tuple to contain the `impact`, etc. The fact that FORWARD's page configuration enables extracting the core data is testimony of the power of a declarative approach fueled by logical statements and queries.

An important piece of data in the core is information about which program was invoked. In the running example, the page invokes the program `save_review` (line 62), and therefore it is important to know upon invocation which one of the many instances of the `save_review` was invoked. There are as many instances as proposals on the page. The core identifies the `proposal_id` for the invoked `save_review`.

FORWARD guarantees that the core data is automatically up-to-date when a program starts its execution. This is a key contribution towards resolving Challenge 3. In a conventional Ajax application, Javascript and Java code has to be written to establish a copy of relevant page data on the server, in a way that they can be subsequently combined with the database.

The `name` attribute convention is reminiscent of the HTML standard's convention to allow a `name` to be associated with each form element and consequently generate request parameters with the provided names. Drawing further the similarities to HTML's request parameters, the *request* objects keep only the tuple of the core that correspond to the invoked program.

Mappings (Section 3.5) raise the level of programming even higher by allowing the developer to select, project and combine data utilizing a mapping interface.

3.5 The Program Configuration

A program configuration is a composition of synchronous *services* in an acyclic structure with a single starting service, which is where the program's execution starts. For example, Figure 2 shows the graphical representation of the `save_review` program configuration, which is composing services `replace_data` and `email`, among others.

Services input data from the unified application state. For example, the starting service `replace_data` takes as input data indicated by the mapping of Figure 4. The invocation of a service has one or more possible *outcomes*, and each outcome (1) generates a corresponding service output, which becomes part of the unified application state, and (2) leads to the invocation of a consequent service or the end of the program. The `replace_data` service has a `success` outcome and a `failure` one. The former adds the `replaced_tuple` to the unified application state and the latter adds a failure `message` in order to facilitate the invocation of the consequent `email` service, which will email the failure message to the application administrator.

FORWARD offers a special service called `page` (depicted by the page icon in Figure 2) that programs use to instruct the FORWARD interpreter of which page to display next.

The transactional semantics of services differentiate the ones that have *side effects* from the ones that do not. A

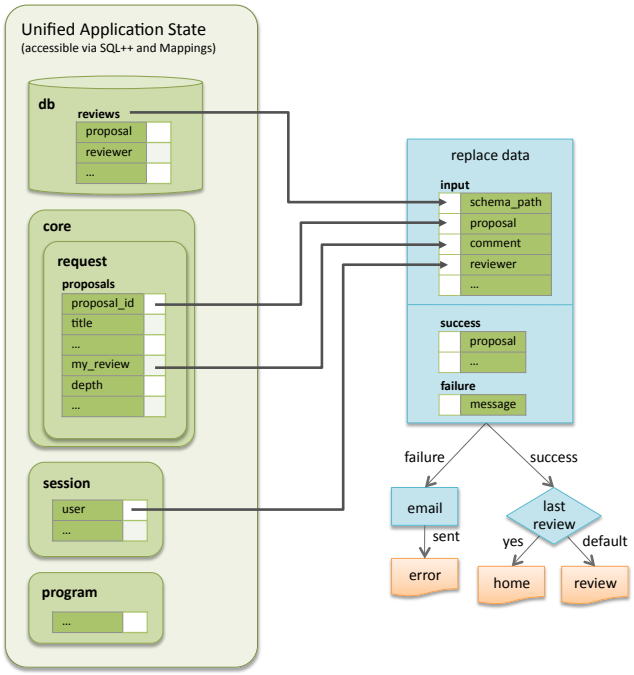


Figure 4: Building the `save_review` program configuration

service has side effects if it makes changes observable by the outside world, be it by updating the unified application state or by invoking an external service, such as sending an email or charging a credit card. The `replace_data` and `email` are examples of services having side effects and are graphically depicted by a rectangle in Figure 2. A service with no side effects is depicted by a rhombus and merely chooses what service will be invoked next. For example, if the outcome of `replace_data` is `success`, then the program will invoke the `last_review` service, which simply checks if the review just saved was the last one to be submitted, and hence does not have any side effects. If so, another `page` service will set the next page to be the `home` page of the application. Otherwise, the next page will be the same as the current `review` page.

Building a program configuration is greatly simplified by SQL++ access to the unified application state. It is further simplified bythe FORWARD *mapping language*. In principle, the service input data can be generated by a SQL++ query. In practice though, the developer rarely does so since the input data of services can be specified much easier using *mappings*. Figure 4 demonstrates how mappings are visually designed inside FORWARD's development environment between the unified application state and the input schema of the `replace_data` service. Notice that the developer seamlessly draws mappings from the persistent database state `db`, the `request` data and the `session` data to the input schema of the service. For consequent services, the developer is also able to draw mappings from the `program` data generated by previous services invocations.

3.6 The Scope of FORWARD Applications

The SQL++ based program and page configurations have limitations, when compared against programs and pages written using combinations of Java, Javascript and SQL. We argue that these limitations are within the spirit of "make easy things easy and difficult things possible": Still every application is doable, while common applications are much easier to write.

In the case of programs, notice that a program is an acyclic graph.Therefore programs comprise (1) sequencing of services, most of which are plain SQL statements, and (2) if-then-else control. However, programs lack explicit loop structures. The restriction is much less severe than it initially appears to be because there are implicit loops in the service implementations. For example, the email service of the example sends an email to each of the many recipients. Nevertheless, the limitation raises three key questions:

First, what percentage of applications do not require explicit loops? Database theorists had introduced the relational transducer [1], which described the program executed after a user interaction by a plain sequence of SQL commands, and essentially conjectured that the business process of most web applications is describable within the expressive power of SQL (notably without recursion). Later, the WAVE project showed that this conjecture applies to well-known web applications, such as `dell.com` [5]. The authors have collected an interesting piece of evidence pointing in the same direction: Two of the authors have given a web application development class, at SUNY Buffalo and UCSD respectively, asking students to specify and build a web application of their choice as a class project. The vast majority of student applications avoid the limitation. For example, in the Winter 2010 UCSD offering of the class, all student projects avoided the limitation.

The next question is how can the limitation be overcome when explicit loops are truly needed in a program? In such cases the developer may write a service in Java and enjoy the full power of Java. Indeed, the developer may write the whole program in Java.

Finally, what are the benefits of the no-loops limitation? The lack of explicit loops simplifies the semantics of service compositions to the point that a program is simply plugging together services via mapping a service's output to other services' input. The practical failure of prior visual programming tools and business process languages that attempted to capture the full extent of programming makes us believe that visual programming should be limited to the simple cases, while Java provides the "bail-out".

4. INTERNAL ARCHITECTURE

The internal architecture of the FORWARD interpreter is illustrated in Figure 5, which displays internal modules (labelled with red numbers) in association with developer-visible concepts of Figure 2. For efficiency of storage and communications, FORWARD maintains a *visual page state*, which provides an abstraction over the browser state by including the externally visible state of visual units, but excluding their implementation details. Two copies of the visual page state lazily mirrored between the browser and the server.

When the user performs interactions such as typing in a text box, the HTML DOM and the JavaScript variables of visual components change in the browser state. The respective *state collectors* of each FORWARD unit synchronize the appropriate part of the browser state with the corresponding part of the *browser-side visual page state*. When the user eventually triggers an event that leads to invoking a

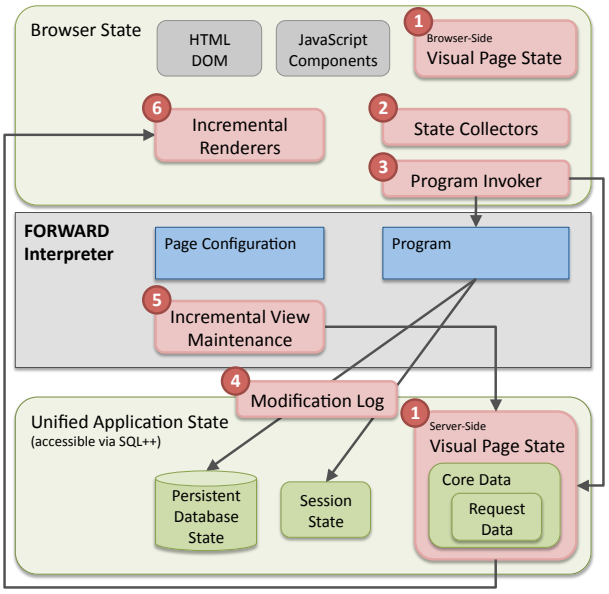


Figure 5: Internal FORWARD Architecture

program, such as clicking the submit button of Figure 1, the *program invoker* guarantees that the browser-side visual page state has been fully mirrored onto the server-side before the program executes. This guarantee is efficiently implemented via incremental writes to the prior visual page state.

Using the program invocation context and page configuration, the interpreter calculates (1) the core data, by projecting only the named attributes of the visual page state, and (2) the request data. As services within the program read from and write to the unified application state, the system also uses a *modification log* to intercept all changes to the unified application state. By using the modification log in combination with the unified application state, the interpreter employs *incremental view maintenance* optimizations to incrementally maintain the current visual page state to the next visual page state [9]. The current implementation uses an off-the-shelf relational database without modification to the database engine. It intercepts changes by instrumenting database-related services. By storing the modification log as well as transient parts of the unified application state in memory-resident tables of the RDBMS, the current implementation issues incremental queries that are more efficient than the original queries in the page configuration, since they retrieve data primarily from the memory-resident tables. As illustrated in [9], incremental view maintenance can speed up the evaluation of page queries by more than an order of magnitude.

Finally, the interpreter uses data diffs to efficiently reflect changes back to the browser-side visual page state. The same data diffs are also provided to the respective *incremental renderers* of each visual unit, which, in turn, programmatically translates the data diff of the visual page state into updates of the underlying DOM elements or method calls of the underlying JavaScript components. Essentially, the incremental renderers modularize and encapsulate the partial update logic necessary to utilize Javascript components, so that developers do not have to provide such custom logic for each page. Also illustrated in [9], in addition to performance gains due to less DOM elements / JavaScript components being initialized, incremental rendering also delivers a better user experience by reducing flicker and preserving unsaved browser state such as focus and scroll positions.

To convert between the different schemas of the core data and the visual page state, the compilation of the page configuration automatically produces a mapping between these two schemas. For example, as discussed in Section 3.2, the core data comprises three (flat) attributes `depth`, `impact` and `overall`, whereas the visual page state represents the corresponding sliders nested within a table unit, thus the mapping language mitigates such structural differences including extraneous attributes (by projection) and nested tuples (by flattening). Mappings are also used to produce type coercions, such as automatically converting an integer attribute in the core data to a float attribute in the visual page state, and vice versa.

5. FUTURE WORK ON OPTIMIZATIONS

As the history of SQL-based systems has shown, a key benefit of declarative approaches is the enablement of (i) automatic optimizations and (ii) static analysis that can improve the operation of the system or detect possible erroneous behaviors. As discussed in Section 4, the efficient incremental maintenance of the Ajax pages is a derivative of the declarative SQL-based approach. FORWARD ongoing and future work will extend the benefits of the declarative approach towards the following optimizations and functionalities, which would otherwise require programmer efforts to be accomplished. Notice that the declarative approach reduces each of the following problems into an extension or specialization of respective problems where the data management community has delivered automatic optimization techniques.

Page query optimization The obvious (according to the semantics) way to collect the data needed by a page configuration is not necessarily the most efficient. For example, the data needed for the page of the running example can be collected by running the outer query of Lines 12-14, which returns proposals, and then, for every proposal tuple, instantiate the `proposal_id` and run the query of Lines 20 to 24, which returns their grades. A more efficient way to collect proposals and grades is by sending a single query. Research in OQL and XQuery has ran into similar issues and has provided a list of rewritings that can be employed for performance gains in such situations.

Pub-sub optimizations for updating myriads of open pages Consider a popular application where there are myriads of pages open at any point in time. When the database data change, these pages have to be correspondingly updated. For example, when a Facebook user changes his status, the currently open pages of his friends must be updated. FORWARD's reduction of the data reported by a page into essentially a rendered view, opens the gate to leveraging database work (e.g., [4, 6]) for the rapid update of only the relevant open pages.

Location transparency, browser side operation and mobility The SQL++ queries over the unified application state are location transparent in the sense that they do not dictate whether they are executed fully at the server or whether they are executed partially at the server and

partially at the browser. While the current version of FORWARD mirrors the browser data on the server and runs SQL++ fully on the server, alternate methods are also possible. A possibility that is beneficial to mobile applications is to create caches of the parts of the application state on the browser. Then certain queries and programs will be able to operate on a disconnected browser.

6. ACKNOWLEDGEMENTS

We thank Kevin Zhao for his worthy-of-authorship contributions. Kevin does not appear in the author list since he is the author of another CIDR 2011 paper. We thank Fernando Gutierrez, Ryan Mourey, Sam Wood and Erick Zamora for the contribution of multiple FORWARD units. We thank Alin Deutsch for many discussions on FORWARD.

7. REFERENCES

[1] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 179–187, New York, NY, USA, 1998. ACM.

[2] Asp.net, 2009. `http://www.asp.net/`.

[3] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[4] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[5] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD Conference*, pages 539–550, 2005.

[6] Y. Diao and M. J. Franklin. Query processing for high-volume xml message brokering. In *VLDB*, pages 261–272, 2003.

[7] Echo web framework, 2009. `http://echo.nextapp.com/site/`.

[8] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with strudel. *VLDB J.*, 9(1):38–55, 2000.

[9] Y. Fu, K. Kowalczykowski, K. W. Ong, K. K. Zhao, and Y. Papakonstantinou. Ajax-based report pages as incrementally rendered views. In *SIGMOD Conference (to appear)*, 2010.

[10] Javaserver pages standard tag library, 2010. `http://java.sun.com/products/jsp/jstl/`.

[11] K. Kowalczykowski, K. W. Ong, K. K. Zhao, A. Deutsch, Y. Papakonstantinou, and M. Petropoulos. Do-it-yourself custom forms-driven workflow applications. In *CIDR*, 2009.

[12] S. Miro. Who moved my state? *Dr. Dobb's*, 2003.

Managing Information Leakage

Steven Euijong Whang and Hector Garcia-Molina
Computer Science Department
Stanford University
353 Serra Mall, Stanford, CA 94305, USA
{swhang, hector}@cs.stanford.edu

ABSTRACT

We explore the problem of managing information leakage by connecting two hitherto disconnected topics: entity resolution (ER) and data privacy (DP). As more of our sensitive data gets exposed to a variety of merchants, health care providers, employers, social sites and so on, there is a higher chance that an adversary can "connect the dots" and piece together our information, leading to even more loss of privacy. For instance, suppose that Alice has a social networking profile with her name and photo and a web homepage containing her name and address. An adversary Eve may be able to link the profile and homepage to connect the photo and address of Alice and thus glean more personal information. The better Eve is at linking the information, the more vulnerable is Alice's privacy. Thus in order to gain DP, one must try to prevent important bits of information being resolved by ER. In this paper, we formalize information leakage and list several challenges both in ER and DP. We also propose using disinformation as a tool for containing information leakage.

1. INTRODUCTION

In this paper we explore the connections between two hitherto disconnected topics: entity resolution and data privacy. In entity resolution (ER), one tries to identify data records that refer to the same real world entity. Matching records are often merged into "composite" records that reflect the aggregate information known about the entity. The goal of data privacy (DP) is to prevent disclosure to third parties of sensitive user (entity) data. For instance, sensitive data can be encrypted to make it hard for a third party to obtain, or the sensitive data can be "modified" (e.g., changing an age 24 to a range "between 20 and 30").

We will argue that in a sense ER and DP are opposites: the better one is at ER, them more one learns about real world entities, including their sensitive information. And to achieve DP, one must try to prevent the bits of information that have been published about an entity from being glued together by ER.

To illustrate, we present a simple motivating example. Consider an entity (person) Alice with the following information: her name is Alice, her address is 123 Main, her phone number is 555, her credit card number is 999, her social security number is 000. We represent Alice's information as the record: { ⟨N, Alice⟩, ⟨A, 123 Main⟩, ⟨P, 555⟩, ⟨C, 999⟩, ⟨S, 000⟩ }. Suppose now that Alice buys something on the Web and gives the vendor a subset of her information, say {⟨N, Alice⟩, ⟨A, 123 Main⟩, ⟨C, 999⟩}. By doing so, Alice has already partially compromised her privacy. We can quantify this "information leakage" in various ways: for instance we can say that the vendor has 3 out of 5 of Alice's attributes, hence the recall is $\frac{3}{5}$. We view leakage as a continuum, not as all-or-nothing. Low leakage (recall in our example metric) is desirable, since the vendor (or third party) knows less about Alice, hence we try to *minimize* leakage. (Note we can actually weight attributes in our leakage computation by their sensitivity.)

Next, say Alice gets a job, so she must give her employer the following data: {⟨N, Alice⟩, ⟨A, 123 Main⟩, ⟨P, 555⟩, ⟨S, 000⟩}. In this case the leakage is $\frac{4}{5}$. This is where ER comes into play: If the employer and vendor somehow pool their data, they may be able to figure out that both records refer to the same entity. In general, in ER there are no unique identifiers: one must analyze the data and see if there is enough evidence. In our example, say the common name and address (and the lack of conflicting information) imply that the records match and are combined (and say the attributes are unioned). Then the third party has increased leakage (recall) to 1.

If Alice wants to prevent this increase in leakage, she may release *disinformation*, e.g., a new record that prevents the resolution that increased leakage. For example, say that Alice somehow gives the vendor the following additional record: {⟨N, Alice⟩, ⟨A, 123 Main⟩, ⟨P, 666⟩, ⟨C, 999⟩}. (Note the incorrect phone number.) Now the vendor resolves this third record with its first record, reaching the conclusion that Alice's phone number is 666. Now, when the vendor and employer pool their information, the different phone numbers lead them to believe that records correspond to different entities, so they are not merged and leakage does not increase. The incorrect phone number also decreases another metric we will consider, precision, since now not all of the third party's data is correct. Thus, leakage can decrease, not just by knowing less about Alice, but by mixing the correct data about Alice with incorrect data.

Before proceeding with the main body of our paper, we

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

highlight the key features of our approach, which we believe make it well suited for studying ER and DP:

- Although not illustrated in our example, our model captures data confidences and multiple attribute values, both which arise naturally in ER. In particular, we believe less information has leaked if a third party is uncertain about Alice's attributes, as opposed to the case where the third party is certain.
- In most DP work, privacy is all-or-nothing, while as mentioned above, our leakage ranges between 0 (no information known by third party) to 1 (all information, and only correct information, is known). We believe that our continuous leakage model is more appropriate in our case: Alice *must* give out some sensitive data in order to buy products, get jobs, and so on. We cannot guarantee full privacy in this context; we can only quantify (and hopefully minimize) leakage. Furthermore, we are able to capture the notion that more leaked attributes is worse than fewer. For example, if a third party only knows our credit card number, that by itself is not a great loss. If the third party also learns our card expiration date, that is a more serious breach. If in addition they know our name and address, the information leakage is more serious.
- So far we have phrased leakage as a bad thing, something to be minimized. However, our model can also be used to study the mirror problem, where a good analyst is using ER to discover information about "bad guys". Here the goal is to maximize leakage, i.e., to discover how to perform ER so we can learn the most correct information about adversaries.

As we will show, our framework can help us answer fundamental questions on information leakage, for instance:

- Alice needs to give certain information to a store. She may want to know the impact of this release: Will the new information allow the store to "connect the dots" and piece together many previously released records? Or will the leakage increase be minimal?
- An analyst may want to understand what ER algorithms are best to increase information leakage.
- Alice may want to use disinformation to reduce the impact of previously leaked information. By adding some bogus information about herself, it becomes more costly for Eve to resolve Alice's correct information.

In this short paper we present a relatively brief summary of our work on the convergence of ER and DP. Our technical report [5] contains full details. In a nutshell, our contributions are two-fold:

(1) We propose a framework for measuring information leakage (summarized in Section 2 of this paper), and

(2) We study how the framework can be used to answer a variety of questions related to leakage and entity resolution. Section 3 of this paper briefly describes two of the questions we have studied (exemplified by the first and third bullets immediately above).

2. MODELS AND ALGORITHMS

We assume a database of records $R = \{r_1, r_2, \ldots, r_n\}$. The database could be a collection of social networking profiles, homepages, or even tweets. Or we can also think of R as a list of customer records of a company. Each record r is a set of attributes, and each attribute consists of a label and value. (In Section 2.4 we extend the model to values with confidences.) We do not assume a fixed schema because records can be from various data sources that use different attributes. As an example, the following record may represent Alice:

$$r = \{\langle \text{N, Alice}\rangle, \langle \text{A}, 20\rangle, \langle \text{A}, 30\rangle, \langle \text{Z}, 94305\rangle\}$$

Each attribute $a \in r$ is surrounded by angle brackets and consists of one label $a.lab$ and one value $a.val$. Notice that there are two ages for Alice. We consider ⟨A, 20⟩ and ⟨A, 30⟩ to be two separate pieces of information, even if they have the same label. Multiple label-value pairs with identical labels can occur when two records combine and the label-value pairs are simply collected. In our example, Alice may have reported her age to be 20 in some case, but 30 in others. (Equivalently, year of birth can be used instead of age.) Although we cannot express the fact that Alice has only one age (either 20 or 30), the confidences we introduce in Section 2.4 can be used to indicate the likelihood of each value.

2.1 Record Leakage

We consider the scenario where Eve has one record r of Alice in her database R. (We consider the case where R contains multiple records in Section 2.2.) In this scenario, we only need to measure the information leaked by r in comparison to the "reference" record p that contains the complete information of Alice. We define $L_r(r, p)$ as the *record leakage* of r against p.

While leakage can be measured in a variety of ways, we believe that the well known concepts of precision and recall (and the corresponding F_1 metric) are very natural for this task. We first define the precision Pr of the record r against the reference p as $\frac{|r \cap p|}{|r|}$. Intuitively, Pr is the fraction of attributes in r that are also correct according to p. Suppose that $p = \{\langle \text{N, Alice}\rangle, \langle \text{A, 20}\rangle, \langle \text{P, 123}\rangle, \langle \text{Z, 94305}\rangle\}$ and $r = \{\langle \text{N, Alice}\rangle, \langle \text{A, 20}\rangle, \langle \text{P, 111}\rangle\}$. Then the precision of r against p is $\frac{2}{3} \approx 0.67$. We next define the recall Re of r against p as $\frac{|r \cap p|}{|p|}$. The recall reflects the fraction of attributes in p that are also found in r. In our example, the recall of r against p is $\frac{2}{4} = 0.5$. We can combine the precision and recall to produce a single metric called $F_1 = \frac{2 \times Pr \times Re}{Pr + Re}$. In our example, the F_1 value is $\frac{2 \times 0.67 \times 0.5}{0.67 + 0.5} \approx 0.57$. It is straightforward to extend our definitions of precision and recall to weighted attributes, where the weight of an attribute reflects its sensitivity.

2.2 Query Leakage

We now consider the case where Eve has a database R containing multiple records. These records can represent information on different entities, and can be obtained directly from the entities, from public sources, or from other organizations Eve pools information with.

When Eve had a single record (previous section), we implicitly assumed that the one record was about Alice and we computed the resulting leakage based on what Eve knew about Alice. Now with multiple records, how does Eve know which records are "about Alice" and leak Alice's information? And what happens if multiple database records are about Alice?

To address these questions, we now define leakage, not as an absolute, but relative to a "query". For instance, Eve

Rec.	Type	Attributes
r_1	Social	⟨N, Alice⟩ , ⟨P, 123⟩ , ⟨B, Jan. 10⟩
r_2	Homepage 1	⟨N, Alice⟩ , ⟨C, Google⟩, ⟨A, 30⟩
r_3	Homepage 2	⟨N, Alice⟩, ⟨E, Stanford⟩ , ⟨A, 20⟩
r_4	Homepage 3	⟨N, Alice⟩, ⟨C, Boggle⟩, ⟨A, 50⟩

Table 1: Records of Alice on the Web

may pose the query "What do I know about {⟨N, Alice⟩, ⟨A, 123Main⟩}". In this case, Eve is saying that the attributes "name: Alice" and "address: 123Main" identify an entity of interest to her, and would like to know what else is known about this entity. Note that this pair of attributes is not necessarily a unique key that identifies entity Alice; the two attributes are simply how Eve thinks of entity Alice. They may be insufficient to uniquely identify Alice, they may be more than is needed. Furthermore, there could be different attributes that also identify Alice.

Our next step is to compute leakage (relative to this query) by figuring out what Eve knows related to {⟨N, Alice⟩, ⟨A, 123Main⟩}. But which database records are related to this query? And how are all the related records combined into what Eve knows about Alice?

To answer these questions, we introduce what we call the *match* and the *merge* functions in ER. A match function M compares two records r and s and returns true if r and s refer to the same real-world entity and false otherwise. A merge function μ takes two matching records r and s and combines them into a single record $\mu(r, s)$.

To illustrate how we use these functions to evaluate leakage, consider the database of Table 1, owned by Eve. Suppose that Eve identifies Alice by the query q={⟨N, Alice⟩, ⟨C, Google⟩} (i.e., the Alice that works at Google). What else does Eve know about this Alice? We use a process called *dipping* to discover database records that match (defined by our function) the query record. That is, we first look for a database record that matches the query. When we find it, we merge the matching record with the query, using our merge function. In our example, say record r_2 matches q, so we obtain $r_q = \mu(q, r_2)$. Then we look for any database record that matches r_q, the expanded query, and merge it to our expanded record. For instance, say $M(r_q, r_1)$ evaluates to true, so we replace r_q by $\mu(r_q, r_1)$. We continue until no other database record matches. This process is called dipping because it is analogous to dipping say a pretzel (the query) into a vat of melted chocolate (the database). Each time we dip the pretzel, more and more chocolate may adhere to the pretzel, resulting in a delicious chocolate-covered pretzel (or an expanded query with all information related to Alice). Note that dipping is a type of entity resolution, where records in the database match against one record (the pretzel), as opposed to any record in the database.

At the end of the dipping process, r_q represents what Eve knows about Alice (q), so we evaluate the leakage by comparing r_q to Alice's private information p, as before. Note that we will get different leakage for different queries. For instance, if Eve thinks of Alice as q={⟨N, Alice⟩}, more records will conglomerate in our example, which may lead to lower leakage (if r_3 and r_4 actually refer to different Alices) or higher leakage (if r_3 and r_4 are the same Alice as r_1 and r_2).

In the remainder of this section we define the dipping process more formally. We start by defining two properties that match and merge functions generally have (and that we assume for our work).

We assume two basic properties for M and μ – commutativity and associativity. Commutativity says that, if r matches s, then s matches r as well. In addition, the merged result of r and s should be identical regardless of the merge order. Associativity says that the merge order is irrelevant.

- Commutativity: $\forall r, s, M(r, s) =$ true if and only if $M(s, r)$ = true, and if $M(r, s) =$ true, $\mu(r, s) = \mu(s, r)$
- Associativity: $\forall r, s, t$, $\mu(r, \mu(s, t)) = \mu(\mu(r, s), t)$

We believe that most match and merge functions will naturally satisfy these properties. Even if they do not, they can easily be modified to satisfy the properties. To illustrate the second point, suppose that commutativity does not hold because $M(r, s)$ only compares r and s if r has an age smaller or equal to s and returns false otherwise. In that case, we can define the new match function $M'(r, s)$ to invoke $M(r, s)$ if r's age is smaller or equal to s's age and invoke $M(s, r)$ if s's age is smaller than r. In the case where the two properties are not satisfied, we only need to add a few more lines in our dipping algorithms for correctness.

Before defining the dipping result of a set of records, we define the "answer sets" for Alice. Throughout the paper, we use the short-hand notation $\mu(S)$ for any associative merge function μ as the merged result of all records in the set of records S (if $S = \{r\}$, $\mu(S) = r$).

DEFINITION 2.1. *Given a query q, a match function M, a merge function μ, and a set of record R, the collection of* answer sets *is $A = \{S_1, \ldots, S_m\}$ where each $S_i \in A$ is a set of records that satisfies the following conditions.*

- $q \in S_i$
- $S_i \subseteq R \cup \{q\}$
- *The records in $S_i - \{q\}$ can be reordered into a sequence $[r_1, \ldots, r_m]$ such that $M(q, r_1) =$ true, $M(\mu(q, r_1), r_2) =$ true, $\ldots$, $M(\mu(\{q, r_1, \ldots, r_{m-1}\}), r_m) =$ true*

For example, suppose we have a database $R = \{r_1, r_2\}$ where r_1 and r_2 are clearly not the same person and have the names Alice and Alicia, respectively. However, say the query q matches either r_1 or r_2 because it contains both names Alice and Alicia and no other information. Then the answer set is $A = \{\{\}, \{q, r_1\}, \{q, r_2\}\}$. Notice that in this example the set $\{q, r_1, r_2\}$ is not in A because r_1, even after it merges with q, does not match r_2.

A dipping result of R is then the merged result of a "maximal" answer set from Definition 2.1 that has no other matching record in R.

DEFINITION 2.2. *Given the collection of answer sets A, a match function M, and a merge function μ, $r_q = \mu(S)$ is a* dipping result *if $S \in A$ and $\forall r \in R - S$, $M(r, \mu(S)) =$ false.*

Continuing our example from above, the dipping result r_q can be either $\mu(r_1, q)$ or $\mu(r_2, q)$ because once q merges with r_1 (r_2), it cannot merge with r_2 (r_1). Notice that we exclude the case of merging multiple records with r_q at a time. For example, even if r_q matches with the merged record $\mu(r, s)$, but not with r or s individually, then we still cannot merge r_q with r and s.

While Definition 2.2 assumes that one record is added to q at a time, it can easily be extended to capture more

sophisticated dipping such as adding multiple records to q at a time.

We define the *query leakage* $L_q(p, q, M, \mu, R)$ of Alice as the maximum value of $L_r(p, r_q)$ for all possible dipping results r_q that can be produced using the match and merge functions M and μ on the database R. In general, deriving the query leakage is an NP-hard problem (see our technical report [5] for a proof).

Properties. We identify two desirable properties for M and μ: representativity and negative representativity. Representativity says that a merged record $\mu(r, s)$ "represents" r and s and matches with all records that match with either r or s. Intuitively, there is no "negative evidence" so merging r and s cannot create evidence that would prevent $\mu(r, s)$ from matching with any record that matches with r or s. It can be shown that representativity guarantees the uniqueness of a dipping result and is needed for efficient dipping. Negative representativity says that two records r and s that do not match will never match even if r or s merges with other records. That is, there is no "positive evidence" where r and s will turn out to be the same entity later on. The negative representativity property also enables efficient dipping. Note that the two properties above do not assume that r matches with s. We can show that none of the properties imply each other.

- Representativity: If $t = \mu(r, s)$, then for any u where $M(r, u) =$ true, we also have $M(t, u) =$ true
- Negative Representativity: If $t = \mu(r, s)$, then for any u where $M(r, u) =$ false, we also have $M(t, u) =$ false

We illustrate a match function called M_c and merge function called μ_u that satisfy both representativity and negative representativity (the proof that M_c and μ_u satisfy the properties and more examples of match and merge functions can be found in our technical report [5]). The M_c function uses a single "key set" for comparing two records. A key set k is a minimal set of attribute labels $\{l_1, \ldots, l_m\}$ that are sufficient to determine if two records are the same using equality checks. All records are assumed to have values for the key-set attributes. The M_c function then matches r and s only if they have the exact same key-set attributes. For example, given the key set $k = \{A, B\}$, the record $r = \{\langle A, a\rangle, \langle B, b\rangle\}$ matches with $s = \{\langle A, a\rangle, \langle B, b\rangle, \langle C, c\rangle\}$, but not with $t = \{\langle A, a\rangle, \langle A, a'\rangle, \langle B, b\rangle\}$. As another example, the record $r = \{\langle A, a_1\rangle, \langle A, a_2\rangle, \langle B, b\rangle\}$ matches with $s = \{\langle A, a_1\rangle, \langle A, a_2\rangle, \langle B, b\rangle\}$, but not with $t = \{\langle A, a_1\rangle, \langle B, b\rangle\}$. The merge function μ_u unions the attributes of r and s (i.e., $\mu(r, s) = r \cup s$). For instance, if $r = \{\langle A, a\rangle, \langle B, b\rangle\}$ and $s = \{\langle A, a\rangle, \langle C, c\rangle\}$, then $\mu(r, s) = \{\langle A, a\rangle, \langle B, b\rangle, \langle C, c\rangle\}$.

Dipping Algorithms. In our technical report [5], we explore various dipping algorithms that exploit properties. (Table 2 in Section 2.5 summarizes their complexities.)

2.3 Database Leakage

What happens if we do not know how Eve identifies Alice, i.e., if we do not have a specific query q? In such a case, we may assume that Eve can think of any one of the records in R as "Alice's record". Thus, for each R record we can compute a leakage number, and by taking the maximum value we can obtain a worst case leakage, representing what Eve can *potentially* know about Alice.

More formally, we define the database leakage $L_d(p, M, \mu, R)$ as $max_{q \in R}\ L_q(p, q, M, \mu, R - \{q\})$. That is, for each record $q \in R$, we compute the dipping of q on $R - \{q\}$ (i.e., the database without q) and choose the worst-case query leakage of Alice as the entire database leakage. In our technical report [5], we explore various algorithms that compute the database leakage of R (Table 2 in Section 2.5 summarizes their complexities) as well as techniques to further scale the algorithms.

2.4 Uncertain Data

As we argued in the introduction, data confidence plays an important role in leakage. For instance, Eve "knows more about Alice" if she is absolutely sure Alice is 50 years old (correct value), as opposed to thinking she might be 50 years old with say 30% confidence, or thinking Alice is either 30 or 50 years old. To capture this intuition, we extend our model to include uncertain data values. Note that there are many ways to model data uncertainty, and our goal here is not to use the most sophisticated model possible. Rather, our goal is to pick a simple uncertainty model that is sufficient for us to study the interaction between uncertainty and information leakage.

Thus, in our extended model, each record r in R consists of a set of attributes, and each attribute contains a label, a value, and a confidence (from 0 to 1) that captures the uncertainty of the attribute (from Eve's point of view). Any attribute that does not exist in r is assumed to have a confidence of 0. As an example, the following record may represent Alice:

$$r = \{\langle \text{N}, \text{Alice}, 1\rangle, \langle \text{A}, 20, 0.5\rangle, \langle \text{A}, 30, 0.4\rangle, \langle \text{Z}, 94305, 0.3\rangle\}$$

That is, Eve is certain about Alice's name and age, but is only 50% confident about Alice being 30 years old, 40% confident in Alice being 30 years old, and 30% confident about Alice's zip code 94305. For each attribute $a \in r$, we can access a's label $a.lab$, a single value $a.val$, and confidence $a.cnf$. In Table 1, the outdated record r_3 of Alice can be viewed to have a lower confidence than the up-to-date record r_2. We assume that attributes in the reference p always have a confidence of 1. We require that no two attributes in the same record can have the same label and value pair.

The confidences within the same record are independent of each other and reflect "alternate worlds" for Eve's belief of the correct Alice information. For example, if we have $r = \{\langle$name, Alice, 1$\rangle$, $\langle$age, 20, 0.5$\rangle$, $\langle$phone, 123, 0.5$\rangle\}$, then there are four possible alternate worlds for r with equal probability: $\{\langle$name, Alice$\rangle\}$, $\{\langle$name, Alice$\rangle$, $\langle$age, 20$\rangle\}$, $\{\langle$name, Alice$\rangle$, $\langle$phone, 123$\rangle\}$, and $\{\langle$name, Alice$\rangle$, $\langle$age, 20$\rangle$, $\langle$phone, 123$\rangle\}$.

We show one example on how our record leakage metrics in Section 2.1 can be extended to use confidences. We first define the notation $IN(r, s)$ for two records r and s as $\{a \in r \mid \exists a' \in s \text{ s.t. } a.lab = a'.lab \wedge a.val = a'.val\}$. That is, $IN(r, s)$ denotes the attributes of r whose label-value pairs exist in s. The precision Pr or a record r against the reference p is defined as $\frac{\Sigma_{t \in IN(r,p)} t.cnf}{\Sigma_{t \in r} t.cnf}$. Compared to the previous definition in Section 2.1, we now sum the confidences of the attributes in r whose label-value pairs are also in p and divide the result by the total confidence of r. The recall Re of r against p is defined as $\frac{\Sigma_{t \in (r \cap p)} t.cnf}{\Sigma_{t \in p} t.cnf}$. This time, we divide the sum of confidences of the attributes in r whose

Confidence	Properties	Query	Database
No	(none)	NP-hard	NP-hard
No	Representativity	$O(N^2)$	$O(N^3)$
No	Neg. Representativity Representativity	$O(N)$	$O(N^2)$
Yes	(none)	NP-hard	NP-hard
Yes	Representativity Monotonicity Increasing	$O(N^2)$	$O(N^3)$
Yes	Neg. Representativity Representativity Monotonicity Increasing	$O(N)$	$O(N^2)$

Table 2: Summary of Leakage Measurements

label-value pairs are also in p by the total confidence of p. We define the record leakage L_r as the F_1 metric $\frac{2\times Pr\times Re}{Pr+Re}$.

In our technical report [5], we elaborate on how to extend our leakage algorithms to take into account confidences. In addition, we discuss two properties (called *monotonicity* and *increasing*) for the extended match and merge functions that can be exploited to compute leakage efficiently.

2.5 Summary of Contributions

As mentioned earlier, Table 2 summarizes the scenarios we have considered in our work. The first column shows whether or not the adversary (which we call Eve) uses confidences in the leakage model. The second column shows properties that are satisfied by the match and merge functions. For each scenario (row) we have developed an algorithm that computes either query or database leakage (given a reference record p for Alice, and a database R held by Eve), and columns 3 and 4 show the complexity of these algorithms. As one can see in the table, the more properties that hold, the more efficiently we can compute leakage.

The properties depend on the data semantics, and different applications (e.g., commercial products, publications, personal information) will have different properties. To show that the properties are achievable in practice, for each scenario in Table 2 we have developed simple match and merge functions that satisfy the corresponding properties. These functions, as well as the algorithms, are all detailed in our technical report [5].

3. USING OUR FRAMEWORK

Our framework can be used to answer a variety of questions, and here we illustrate two questions. As we use our framework, it is important to keep in mind "who knows what". In particular, if Alice is studying leakage of her information (as in the two examples we present here), she needs to make assumptions as to what her adversary Eve knows (database R) and how she operates (the match and merge functions, and dipping algorithm Eve uses). These types of assumptions are common in privacy work, where one must guess the sophistication and compute power of an adversary. On the other hand, if Eve is studying leakage she will not have Alice's reference information p. However, she may use a "training data set" for known individuals in order to tune her dipping algorithms, or say estimate how much she really knows about Alice.

3.1 Releasing Critical Information

Suppose that Alice wants to purchase a cellphone app from one of two stores S_1 and S_2, and is wondering which purchase will lead to a more significant loss of privacy. Both stores require Alice to submit her name, credit card number, and phone number for the app. However, due to Alice's previous purchases, each store has different information about Alice. In particular:

- Alice's reference information is $p = \{\langle \text{N}, n_1, 1\rangle, \langle \text{C}, c_1, 1\rangle, \langle \text{C}, c_2, 1\rangle, \langle \text{P}, p_1, 1\rangle, \langle \text{A}, a_1, 1\rangle\}$ where N stands for name, C for credit card number, P for phone, and A for address.
- Store S_1 has one previous record $R_1 = \{r = \{\langle \text{N}, n_1, 1\rangle, \langle \text{C}, c_1, 1\rangle, \langle \text{A}, a_1, 1\rangle\}\}$. That is, Alice bought an item using her credit card number and shipping address. (We omit the item information in any record for brevity.)
- Store S_2 has two previous records $R_2 = \{s = \{\langle \text{N}, n_1, 1\rangle, \langle \text{C}, c_1, 1\rangle, \langle \text{P}, p_1, 1\rangle\}, t = \{\langle \text{N}, n_1, 1\rangle, \langle \text{C}, c_2, 1\rangle, \langle \text{A}, a_1, 1\rangle\}\}$. Here, Alice has bought items using different credit cards. The item of s could be a ringtone that required a phone number for purchasing, but not a shipping address.
- Both S_1 and S_2 require the information $u = \{\langle \text{N}, n_1, 1\rangle, \langle \text{C}, c_2, 1\rangle, \langle \text{P}, p_1, 1\rangle\}$ for the cellphone app purchase. Since Alice is purchasing an app, again no shipping address is required.

To compute leakages, say Alice is only concerned with the previously released information, so she assumes that the database at store S_1 only contains record r, while the database at store S_2 only contains s and t. (The stores are not colluding in this example.) Alice also assumes that two records match if their names and credit card numbers are the same or their names and phone numbers are the same, and that merging records simply performs a union of attributes.

Under these assumptions, before Alice's app purchase, the database leakage for both stores is $\frac{3}{4}$. For the first store, R_1 only contains one record r, so the database leakage is $L_r(p,r) = \frac{2\times 1\times 3/5}{1+3/5} = \frac{3}{4}$. For the second store, R_2 contains two records s and t, so we need to take the maximum of the query leakages of s and t. Since s and t do not match with each other (i.e., they do not have the same name and credit card or name and phone combination), the dipping result of s is s while the dipping result of t is t. Hence, the database leakage is $max\{L_r(p,s), L_r(p,t)\} = max\{\frac{3}{4}, \frac{3}{4}\} = \frac{3}{4}$.

If Alice buys her app from S_1, then its database will contain two records, r and u. In this case, the database leakage at S_1 is still $\frac{3}{4}$ because r and u do not match and thus have the same query leakage $\frac{3}{4}$ (the maximum query leakage is thus $\frac{3}{4}$). On the other hand, if Alice buys from S_2, the database at S_2 will contain s, t, and u. Since u matches with both s and t, the dipping result of u is $\mu(\{s,t,u\})$, which is identical to p. Hence, the database leakage becomes 1.

To compare Alice's two choices, it is useful to think of the *incremental leakage*, that is, the change in leakage due to the app purchase. In our example, the incremental leakage at S_1 is $\frac{3}{4} - \frac{3}{4} = 0$ while the incremental leakage at S_2 is $1 - \frac{3}{4} = \frac{1}{4}$. Thus, in this case Alice should buy her app from S_1 because it preserves more of her privacy.

3.2 Releasing Disinformation

Given previously released information R, a match function M, and a merge function μ, Alice may want to release

either a single record or multiple records that can decrease the query or database leakage. We call records that are used to decrease the database leakage *disinformation* [1] records. Of course, Alice can reduce the query or database leakage by releasing arbitrarily large disinformation. However, disinformation itself has a cost. For instance, adding a new social network profile would require the cost for registering information. As another example, longer records could require more cost and effort to construct. We use $C(r)$ to denote the entire cost of creating r.

We define the problem of minimizing the database leakage using one or more disinformation records. Given a set of disinformation records S and a maximum budget of C_{max}, the optimal disinformation problem can be stated as the minimization function presented below:

$$\begin{array}{ll} \text{minimize} & L_d(p, M, \mu, R \cup S) \\ \text{subject to} & \Sigma_{r \in S} C(r) \leq C_{max} \end{array}$$

The problem of minimizing the query leakage can also be stated by replacing L_d by L_q in the above formula. The set of records S that minimizes the database leakage within our cost budget C_{max} is called "optimal" disinformation.

A disinformation record r_d can reduce the database leakage in two ways. First, r_d can perform *self disinformation* by directly adding the irrelevant information it contains to the dipping result r_q that yields the maximum query leakage. For example, given the database $R = \{r, s, t\}$ and a reference p, suppose the database leakage is $L_r(p, \mu(r, s))$. Then r_q can be created to match with $\mu(r, s)$ and to contain bogus data not found in p, thus decreasing database leakage to $L_r(p, \mu(\{r, s, r_d\}))$. Second, r_d can perform *linkage disinformation* by linking irrelevant records in R to r_q. For example, say that t contains totally irrelevant information of the target p. If r_d can be made to match with both $\mu(r, s)$ and t, then the database leakage could decrease to $L_r(p, \mu(\{r, s, t, r_d\}))$ because of r_d. Of course, r_d can also use both self and linkage disinformation.

When creating a record, we use a user-defined function called $Create(S, L)$ that creates a new minimal record that has a size less or equal to L and is guaranteed to match all the records in the set S. If there is no record r such that $|r| \leq L$ and all records in S match with r, the $Create$ function returns the empty record $\{\}$. A reasonable assumption is that the size of the record produced by $Create$ is proportional to $|S|$ when $L > |S|$. We also assume a function called $Add(r)$ that appends a new attribute to r. The new attribute should be "incorrect but believable" (i.e., bogus) information. We assume that if two records r and s match, they will still match even if Add appends bogus attributes to either r or s. (Notice that this property is similar to the representativity property for match and merge functions.) The $Create$ function is assumed to have a time complexity of $O(|S|)$ while the Add function $O(|r|)$. In our technical report [5], we list more details to consider when adding bogus attributes to r_d using the function Add.

We propose disinformation algorithms in our technical report [5], both for the case we release a single disinformation record (S is size 1) and the case where we release multiple records. In addition, we consider scenarios where different properties hold. If both representativity and negative representativity hold, one can show that a new record can only use self-disinformation to lower the database leakage, which enables efficient algorithms that return the optimal disinformation. If the properties do not hold, a new record can also use linkage disinformation, and we might have to consider all possible combinations of irrelevant records in the worst case (which makes the disinformation problem NP-hard). Thus, we also propose a heuristic algorithm that searches a smaller space, where we either combine two irrelevant records and use self disinformation or use self disinformation only. As more properties are satisfied by the match and merge functions, the more efficient the disinformation algorithms become. Similar results can be obtained when using confidences and the monotonicity and increasing properties for the extended match and merge functions.

[1] A classic example of disinformation occurred before the Normandy landings during World War II where British intelligence convinced the German Armed Forces that a much larger invasion was about to cross the English Channel from Kent, England.

4. RELATED WORK

Many works have proposed privacy schemes for data publishing in the context of linkage attacks. Various models including k-anonymity [3] and l-diversity [1] guarantee that linkage attacks on certain attributes cannot succeed. In contrast, we assume that the data is already published and that we want to manage the leakage of sensitive information.

Several closely related products manage information leakage. A service called ReputationDefender [2] manages the reputation of individuals, e.g., making sure a person's correct information appears on top search results. TrackMeNot [4] is a browser extension that helps protect web searchers from surveillance and data-profiling by search engines using noise and obfuscation. In comparison, our work complements the above works by formalizing information leakage and proposing disinformation as a general tool for containing leakage.

5. CONCLUSION

We have proposed a framework for managing information leakage and studied how the framework can be used to answer a variety of questions related to ER and DP. The algorithms for computing leakage become more efficient as the match and merge functions satisfy more properties. We have studied the problems of measuring the incremental leakage of critical information and using disinformation as a tool for containing information leakage. We believe our techniques are preliminary steps to the final goal of truly managing public data, and that many interesting problems remain to be solved.

6. REFERENCES

[1] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, page 24, 2006.

[2] ReputationDefender. http://www.reputationdefender.com.

[3] L. Sweeney. k-anonymity: A model for protecting privacy. *IJUFKS*, 10(5):557–570, 2002.

[4] TrackMeNot. http://cs.nyu.edu/trackmenot.

[5] S. E. Whang and H. Garcia-Molina. Managing information leakage. Technical report, Stanford University, available at http://ilpubs.stanford.edu:8090/983/.

eXO: Decentralized Autonomous Scalable Social Networking

Andreas Loupasakis
Computer Engineering & Informatics Dept.
University of Patras, Greece
loupasak@ceid.upatras.gr

Nikos Ntarmos
Computer Science Dept.
University of Ioannina, Greece
ntarmos@cs.uoi.gr

Peter Triantafillou
Computer Engineering & Informatics Dept.
University of Patras, Greece
peter@ceid.upatras.gr

ABSTRACT

Social networks have been receiving increasingly greater attention. As they stand now, users are required to upload their content to make it available to others. Typically, this involves releasing ownership and control. As a result, battles have begun regarding the ownership, exploitation, and control of content between users and social network owners. Further, given the rates of growth witnessed recently, questions about the scalability of the social network services are being raised. Therefore, the following questions naturally emerge: Is it possible to architect, design, and implement decentralized social networking services that ensure scalability and efficiency, while, respecting users' control of their content? Can this be achieved while content can be available to others for viewing and commenting, and permit key social networking activities like tagging? This paper presents *eXO*, a completely decentralized, scalable system that offers fundamental social networking services. We describe the architecture of *eXO* and its key components which encompass (i) techniques for content indexing, (ii) novel algorithms for ranked retrieval, (iii) appropriate similarity definitions that consist of a 'content' and a 'social' part, (iv) novel algorithms for efficient distributed content retrieval, (v) novel techniques that efficiently and scalably facilitate tagging and exploit tags to enrich query results, and (vi) novel scalable methods which permit the creation of personal networks, which allow for more "intimate" sharing and associations between users. We report on our evaluation which showcases its efficiency and scalability characteristics. *eXO* source code will be freely available to all, to use and test.

Categories and Subject Descriptors

H.3.4 [**Systems & Software**]: Distributed Systems, Query Processing

General Terms

Algorithms, Design, Performance

5th Biennial Conference on Innovative Data Systems Research (CIDR'11) January 9–12, 2011, Asilomar, California, USA

Keywords

Social networks, peer-to-peer, tagging, similarity, top-k

1. INTRODUCTION

Social networking services and entrepreneurship have been growing steadily and rapidly. User-generated content (mainly multimedia, such as images, videos, etc) is being produced at high rates. Interestingly, people show a great desire to share their content, view content shared by others, tag (comment on) it, search for content of interest and/or other users with specific profiles, and make 'friends' (become members of specialized groups sharing common interests). This is not all new. Since Napster appeared, a large number of content sharing applications have been implemented, providing a variety of options and actions to the user. Nowadays, users have moved on, creating large scale social network applications, such as Facebook, MySpace, YouTube, Flickr, etc. These applications run on centralized sites facilitating different kinds of social network sharing. The heart of these applications are users' interactions: content sharing and viewing experiences are being enriched, taking into account the profiles of those contributing the content and of those who have tagged it.

As they now stand, social network services force users to upload their content to a specific site in order to make it available to others. Typically, this involves releasing ownership/control of uploaded content. After a few years of tremendous growth in popularity and use, already some issues have been raised and battles have begun regarding the ownership and exploitation of content between users and social network site owners. Further, given the rates of growth witnessed recently, the scalability of the social network services becomes increasingly doubtful. Therefore, the following questions naturally emerge: Is it possible to architect, design, and implement decentralized social networking services that ensure scalability and efficiency? In addition, can users be allowed to retain full control of their content and efficiently make it available to others for viewing, sharing, and commenting? And, all this, while permitting other key social networking activities like tagging, so to enrich the users' querying experiences?

Our interest in future social network services is to ensure two key characteristics: highly decentralized social network functionality and with full user control when sharing. First, we wish to "go distributed". Already our community has produced a great wealth of knowledge for highly decentralized, efficient, and scalable content sharing – see for exam-

ple P2P network architectures (such as those built on top of DHTs). We wish to build upon these efforts and answer the questions whether the scalability of the underlying network can be leveraged to ensure scalable social network sharing and accessing and whether this can be done efficiently. Second, so far content providers may lose control over their data. We believe it is important for users to maintain control over their data: for instance, they should be able to control who accesses their data, when and how their content is replicated, if at all, and to which sites, etc.

These goals bear a number of important implications for system design which, in turn, create new research challenges. First, to respect autonomy, content must not be stored remotely. Further, to facilitate sharing, this content must be indexed appropriately so that users can search for it and locate it. This index must be distributed for scalability and efficiency reasons. Thus, no central organization/site exists, which is responsible for the content and related metadata, such as indices. Moreover, great care must be exercised with respect to how and what will be inserted into this distributed index, in order to avoid scalability and efficiency barriers, as has been shown in the realm of P2P search engines [14]. Further, defining appropriate similarity functions must account for high expense when computing relevant statistical metadata in distributed settings (despite valiant efforts [2]).

Second, autonomy implies for users the ability to name content on their own, resulting in a duplicity of names for the same content. Thus, any particular *search query* may be matched to a number of different sources. Our system must on the one hand efficiently maintain such metadata on data sources, and on the other, provide efficient algorithms for retrieving this content.

Third, the semantics of search queries are different. A query can request one such item, or all of them, or (most likely) a specific number of items, e.g., running top-k queries [10]. The ranking of query results must of course take into account the characteristics of content items and how close they are to a given query (a la traditional IR environments). However, in addition, ranking must take into account other social-network characteristics, such as the profile of the querying user, the profiles of the users who uploaded content items, and their similarity. Thus, new similarity functions are needed to facilitate such query-result ranking.

Fourth, running distributed top-k queries is a notoriously difficult problem. Despite recent advances (for instance [6, 15]), distributed top-k execution can introduce a rather large cost for the overall system resources (such as number of messages, network bandwidth, and local per-peer processing costs) and for the user (large query execution times). As we shall see, this cost depends on the similarity function. It is therefore imperative to produce similarity functions that can both exploit social network information and facilitate efficient distributed top-k query execution.

Finally, dealing with and exploiting user tags in a decentralized environment is not simple. On the one hand, it is important for the system to exploit these tags in order to identify relevant content [8, 13]. However, on the other, given that tagging is a very popular activity, this must be done in a way that does not introduce great overheads and scalability problems. Reconciling these goals is a key undertaking for any decentralized social network system.

Diaspora[1], an open-sourced personal web server implementing decentralized social networking services, is a first commercial crack at this; the $200k in donations received by the Diaspora authors and their recent (albeit early alpha-stage) source code release, testify to the plausibility of and widespread interest in such systems. Diaspora only implements the most basic of social networking primitives (that is, friends and friend groups), lacking any support for more advanced operations, such as (even simple) system-wide content/user queries – let alone such features as distributed top-k operations, tag clouds, personal social networks, etc.

With this work we present *eXO*, a novel decentralized system offering fundamental social networking services, consisting of:

- mechanisms for indexing user-generated content and related metadata definitions;
- appropriate similarity functions that combine social networking features with traditional query-to-content relevance;
- discovery and retrieval of related content via a combination of DHT-based indexing and of unstructured, query-relevant, social networks,
- efficient top-k algorithms for search-for-content and search-for-user queries;
- efficient algorithms for retrieving content items exploiting the architecture of the overlay network on the one hand, and the characteristics of content in social network sharing environments on the other;
- scalable social tagging mechanisms and methods which can exploit social tags to improve the quality of query results;
- methods for building social networks; and
- an implementation of the system and a performance evaluation substantiating the claims for scalability and efficiency.

To our knowledge, this is the first effort addressing comprehensively the design and architecture of decentralized social network services. The rest of this paper is organized as follows. First we present a brief overview of DHT basics. Section 2 discusses the data and query models, and section 3 the system architecture and key design decisions. Section 4 discusses query processing chores, presenting our approach for indexing, similarity definitions, ranking of query results, and optimized algorithms for retrieving content over the DHT. Section 5 discusses how we facilitate tagging and how we exploit it for search and retrieval. Section 6 shows how to build unstructured (personal) social networks that can be used to improve search results quality. Section 7 discusses our implementation and experimental results. Section 8 discusses related work and finally sections 9 and 10 conclude this work.

2. THE DATA AND QUERY MODEL

eXO is built upon a network, consisting of a possibly large number of nodes. Each node runs a routing protocol for a structured overlay DHT network (such as Pastry [9], Chord [21], Tapestry [24], CAN [18], etc.) Shareable content and nodes acquire IDs from the same ID space, using a public hash function, such as SHA-1, matching each of them to a specific position in the ID space. Structured overlay networks require nodes to maintain routing tables of size

[1] `http://joindiaspora.com/`

logarithmic to the size of the network (i.e., number of participating nodes). Further, they ensure that a message will be delivered to its destination in a logarithmic (again, relative to the size of the network) number of hops. For instance, in Pastry[9], the protocols guarantee $O(\log_{2^b} N)$ hops for the routing process, where N is the number of nodes in the network and typically $b = 4$. In Pastry, this routing process will also end up pointing to the node with the arithmetically closest ID to the destination ID (the input ID of routing function).

In *eXO* a user is associated with a unique network ID (*UID*), which is computed using a hash function (e.g. SHA-1) on a user-specified string, e.g. an e-mail address. Each user is connected to a specific node of the network; thus, a UID plays indirectly the role of a node identifier as well. Last, we use the term *content* to refer to every type of data that can be indexed, retrieved, and stored in the system (in *eXO* we are mostly interested in images, audio, and video content and secondarily in text).

Content & User Profiles. For indexing purposes, each content item is represented by a set of terms (keywords) that describe it. We call this set of terms the *content profile*. Similarly, each user is described by a set of terms defined by the users themselves, denoted the *user profile*. Each term is associated with an ID (term ID – *TID*). TIDs are also computed using a hash function on the term string. For example, a content profile term could be the file name of some multimedia file or words taken from the id3 tag of an mp3 file, while a user profile term may include information about user interests, *expertise* in some area, etc. A checksum of the actual content or user ID is computed to distinguish among the many different objects with the same profile. Content items may be replicated from the source to adjacent nodes in the ID space of the overlay in order to improve content availability. Content is indexed to appropriate network nodes, so it can be discovered. Note that when a content item is first shared, its owner decides on the content's profile - that is, the set of terms with which to index it. In any case, the owner of a content item holds complete power over what tags are attached to her profile/content items; however, subsequent tagging by other users (to be discussed shortly) may sway her selection of index tags and lead to a richer and more accurate indexing. Moreover, user profile terms play a central role during the ranking of query answers: for example, items uploaded by users who are either experts in a related area or whose profile matches a specific input user profile, are ranked higher than content items uploaded by other users.

Tags. *Tags* are terms contributed by users to describe a specific content item or user. Tags are very important in order to help achieve higher query result quality by exploiting the community's wisdom. As above, a tag term is associated with a TID. In *eXO* tags are stored at the *taggee*'s side associated with the tagged content and/or user profile. In addition, inverted lists for each tag term are maintained at the *tagger*'s node, associating each such term with a set of thus tagged resources. This information can be exploited appropriately to enhance the searching process as we will explain later. We refer to *social tags* as the tags whose tagger is a user other than the owner of the tagged item, as opposed to *self (owners') tags*.

Friends. Friendship is a major aspect in any social network service environment. A user can make friends and share more private data with them. To support this functionality two users can become *friends* by a typical handshake process, in which the first user requests a friendship with another user and the latter can accept or reject this request. The system stores *friend lists* (that is, list of friend UIDs) on the related nodes, updated every time a friend is added or deleted. The friend list structure is used to locate friends in the network, to allow access control of the data and to improve the lookup ranking process. The latter can be achieved by ranking the friends' content higher.

Queries. Queries in our model refer to content or user profiles indexed in the network. A query is a set of keywords and the query processing engine is meant to return the top-k most relevant items to the query, using a data structured called the "Catalogue" (to be discussed shortly). A query is a set of keywords which are used to obtain a number of data sources for the most similar to the keywords catalogue entries, supported through a similarity matching and a top-k ranking process. To be more formal, let $U = \{u_1, ..., u_m\}$, $C = \{c_1, ..., c_n\}$, $T = \{t_1, ..., t_o\}$ be the set of all user profiles, the set of all content profiles, and the set of all terms, respectively. For example, $u_i = \{t_i, ..., t_a\} \in U$ is a single user profile, and $c_i = \{t_j, ..., t_b\} \in C$ is a single content profile. A query may contain a content part and a user part and is given as $Q = Q_c + Q_u$, where $Q_c = \{t_1, ..., t_{q_1}\}$ is the content part and $Q_u = \{t_1, ..., t_{q_2}\}$ is the user profile part. In brief, queries may contain both or either of a content-related part and a user-related part, where the content (user) part may be missing when the querying party is specifically interested in finding user (respectively, content) profiles.

If we search for users, then a user query is formed. In this case $Q_u \neq \emptyset$, consisting of the set of terms which are used to route to the catalogues. Q_c is generally ignored in this case. User queries are used to list the k most similar user catalogue entries to the querying terms. The returned user catalogue entries contain user profiles and are also indices of the users' source nodes, so they can be used to navigate towards the user nodes in order to browse or tag their content.

On the other hand, content queries are used to find catalogue entries of shared content objects. It this case, $Q_c \neq \emptyset$ is used to route the process to the appropriate catalogue nodes. If, in addition, $Q_u \neq \emptyset$ then Q_u is used to boost the scoring of content contributed by similar users in the catalogues. The content profiles are encapsulated in content catalogue entries together with the content owner's source node; thus, in essence these catalogue entries are overlay "links" to access the shared content.

3. EXO ARCHITECTURE AND DESIGN

Here we discuss the basic components of the system and its design rationale.

Autonomy and privacy. The system is designed so that users can exercise full control over their content and their node's resources. In *eXO*, content shared by a user is kept only on the user's node (source node). For content availability reasons (e.g. for when the user leaves the network due to either a failure or a graceful disconnection) and at the user's discretion, the system can further maintain a number

CATALOGUE FOR "ACROPOLIS" @ node *hash(Acropolis)*		
UID	CONTENT PROFILE	USER PROFILE
A1FE458B...	{term1, ..., Acropolis, ..., termm}	{term2 , 'peter'. term r}
....	....	....

Figure 1: Example catalogue data

of content replicas. These replicas are stored only on nodes adjacent to the user node in the ID space, and their maintenance is performed automatically by the substrate network (e.g Pastry[9]).

The user can mark her content as *public* or *private* to define which users can access it. Public content items are indexed in the network and can be seen by anyone in the overlay network, while private content is not indexed and is thus non-visible to the public. Instead, it is made visible and can be accessed only by friends. Similarly, there exists both a *public* and a *private* user profile in the network; the public user profile is also indexed and may be replicated, while the private part is stored only on the source node. Note that owners can reject access requests made by other users, even for public content. The basis for this rejection can be any criteria the owner chooses to use (e.g., on the id of the querying user).

Node roles. The software running on each node takes on a number of tasks. First, it acts as a front end, serving user requests and dispatching them to the appropriate peer nodes in the system (*request resolver role*). Second, it provides a *network storage* interface for the content and profile replicas. Last, nodes may store indexing data structures for remote shareable content and user profiles in the system, in which case they are denoted *Catalogue Nodes*.

Catalogues. Each term ID is assigned to such a node (using the DHT's mapping primitive); that node then stores, for every TID it is responsible for, a data structure coined the *Catalogue*, mapping the TID to a set of related UIDs, along with (a portion of) the user profile, content profiles, and optionally other data (e.g. friend lists). Thus, every TID in the network has its own Catalogue data structure, keeping track of user nodes storing shareable content or user profiles containing the TID keyword.

A Catalogue has two parts. First, the User part is used to store the user profile and the UID of each user whose public profile contains the relevant term; the user catalogue is used to answer user-centered requests, such as "find the top-k similar users". Second, the Content part (see figure 1) holds the content profile terms and the user profile of the content owner, mapped to the content owner UID. This data structure is used to answer content related queries such as "find the top-k similar content items". As an enhancement for similarity matching, we also store the corresponding owner's user profile in each content catalogue entry. This allows to boost our scoring mechanism by using the comparison of the owners' profiles with the querying user profile.

Handling Tags. Social tags are stored only on the tagger and taggee nodes associated with the annotated object. We chose not to index tags in the network for that would introduce great overhead costs [11]. Indexing tags, on the other hand, would provide extra scoring capabilities, e.g. similarity boosted by tags. To reconcile this trade-off, we allow the benefits of social tagging for query resolution as follows: we store locally social tags (avoiding their distribution and related maintenance overheads) and we use a low cost mechanism based on query reformulation to enrich our search results with tag data (to be described shortly).

The public network and Personal Social Networks. The above rationale leads to a two-pronged approach. The *public network* consists of the DHT and the content and user profiles which are stored at the DHT nodes and have been made DHT-indexable and DHT-accessible. On this public network, users can issue queries specifying user profiles of interest and thus identifying other interesting users. Accessing those users' nodes, a querying user can also identify and retrieve tags with which the interesting user profiles have been annotated. Then, a querying user can expand her query and thus identify more interesting users. Once identified, the querying user can add them to her friends' lists, establishing and enriching her personal social network.

Thus, *eXO* consists of a public, structured network and a series of private/personal, unstructured networks. Owners have complete autonomy of what content to share, who to befriend, what tags to accept, what further functionalities to perform (e.g., transitively expanding friendship relationships) and exercise it at the per-user level, at the per-content item level, etc.

4. QUERY PROCESSING

This section describes the indexing and query execution processes, similarity definitions, the mechanism for ranking query results, and content retrieval optimizations.

Indexing and Catalogues. Indexing is performed as follows. First, the object's (user or content) profile is fetched. For each term of the profile, the TID is computed and is used as input to the overlay routing function. An indexing message is created containing the UID of the source node, the object's profile, plus, for content objects, the owner's user profile. The message is routed towards the destination (which is the overlay node responsible for TID). This destination node plays the role of the catalogue node for this term. Depending on the type of object (user or content profile), an appropriate catalogue entry is formed from the message data and stored in the catalogue. The same steps are taken *in parallel* for every term in the object's profile. For content objects or users which have very large profiles, indexing of all the terms would be very network-hungry. For this reason the indexing process may choose not to use all of the terms in the profile. Here, we must note that real social data, as shown in section 5 by our crawling efforts, usually contain user or content profiles of acceptable size.

Similarity and Local Ranking. Search query messages are delivered to every catalogue node corresponding to the query terms' TIDs. There, a *local* similarity matching process takes place to compare the catalogue entries with the query terms and then a *local* ranking of the catalogue entries is performed. This similarity function has a user and a content part; the former is matched against terms in content profiles while the latter against terms in user profiles. This allows us to search for content with specific properties, users

with specific properties, or a combination of these two (i.e. content with specific properties shared by users with specific properties). To compare sets of terms of user profiles and/or content profiles we use the vector space model. Specifically, we consider a multidimensional space, where every discrete term determines just one dimension. Vectors of term weights are used to map profiles to this multidimensional space, producing feature vectors. Vectors are compared by using the cosine similarity equation, with similar profiles having high cosine values.

A significant part of this scoring computation is the term weight formula. Firstly, let U, T, C be the set of all user profiles, the set of all terms and all content profiles, respectively. $u \in U$ is a single user profile, $c \in C$ is a single content profile and $t \in T$ is a single term. Obviously, $T = (\cup_{i=1}^{|U|} u_i) \bigcup (\cup_{j=1}^{|C|} c_j)$. In our approach weight w_{t_i} of term t_i, $0 < i \leq |T|$ is computed by the next general approach reflecting the relations among content, user profile, and term:

$$w_{t_i} = a_c w_{t_i}^c + a_u w_{t_i}^u$$

where $w_{t_i} \in [0, 1]$, $w_{t_i}^c$ ($w_{t_i}^u$) denotes the relevant weight of the content (respectively user) profiles for t_i, and a_c and a_u are coefficients which indicate the significance of each weight, with $a_c + a_u = 1$.

We adopt a simplified boolean weight formula to compute the weights; that is, $w_{t_i}^c, w_{t_i}^u \in \{0, 1\}$, where a value of 1 signifies that the term t_i exists in the user/content profile. Let $E = [w_{t_1} w_{t_2} ... w_{t_m}] = E_c + E_u$ be the vector of the catalogue entry's weights for the corresponding content profile $c_e = t_1, t_2, ..., t_{m_c}$ and user profile $u_e = t_1, t_2, ..., t_{m_u}$, where m denotes the number of unique terms in the catalogue entry ($|u_e \cup c_e|$ of this entry), E_c is the vector of content profile weights and E_u is the vector of user profile weights. Remember that content catalogue entries contain both a content profile part and a user profile part, while user catalogue entries contain only a user profile part (i.e., for user catalogue entries, it holds that $E_c = \emptyset$).

Consider the query vector $Q = [w_{t_1} w_{t_2} ... w_{t_l}] = a_c Q_c + a_u Q_u$ for the corresponding user profile $u_q = t_{1_u}, t_{2_u}, ..., t_{l_u}$ and content profile $c_q = t_{1_c}, t_{2_c}, ..., t_{l_c}$, where l is the number of unique query terms, and Q_c, Q_u are the vectors corresponding to E_c, E_u, respectively. The weights of the query vector are computed by the same formulas as the catalogue entries. Using the cosine similarity as scoring function we get:

$$\text{Sim}[Q, E] = \frac{Q \otimes E}{\|Q\| \|E\|} \tag{1}$$

The $Q \otimes E$ is the inner product of the two vectors and $\|Q\| \|E\|$ is the product of their norms. The denominator is used for normalization purposes, in order to use unitary vectors.

Using the previous weights and according to the query model, we discern two cases.

1. Queries on user profiles. In this case only the user profile part contributes to the score computation, thus $w_{t_i} = w_{t_i}^u, a_u = 1, a_c = 0$, and:

$$\text{Sim}[Q, E] = \frac{Q_u \otimes E_u}{\|Q_u\| \|E_u\|} = \frac{|u_q \bigcap u_e|}{\|Q_u\| \|E_u\|} \tag{2}$$

2. Queries on content profiles. In this case we allow the similarity between the sharing and the querying users' profiles to be taken into account, thus $w_{t_i} = \frac{1}{2} w_{t_i}^c + \frac{1}{2} w_{t_i}^u, a_c = 1/2, a_u = 1/2$, and:

$$\text{Sim}[Q, E] = \frac{Q_c \otimes E_c + Q_u \otimes E_u}{\|Q\| \|E\|} = \frac{|u_q \bigcap u_e| + |c_q \bigcap c_e|}{2 \cdot \|Q\| \|E\|} \tag{3}$$

A catalogue node receiving a search request, uses the above formulas to score each catalogue entry, creating a sorted score table. Note that the above can straightforwardly be enriched with more elaborate social metadata, such as friends and measures of friendship [19, 20] at least for "direct" (non-transitive) friends.

Global Ranking. After the similarity processes have ended in each of the catalogue nodes and a sorted list of catalogue entries has been created, the appropriate results must be returned to the querying node. The search mechanism considers a similarity threshold on the obtained results. This implies a top-k choice of local catalogue entries. As we will explain and prove in the next section, our scoring process has each catalogue node compute and return a score that is a *global score* for the object represented by the entry for a query. So it is enough to get only the first k results from each local sorted list of catalogue entries and send them to the query origin node in order to get the global top-k list. This phase is called *global ranking*. Every catalogue node returns the top-k catalogue entries from the local sorted list along with their corresponding score values to the query source node. The latter then merges all incoming lists and a new list of catalogue entries is formed, sorted by score, from which the global top-k result list emerges.

Note that whenever an item is found in more than one catalogue node lists, its score is *not* aggregated (summed). This is so, because each catalogue entry already reflects a global relevance score. Recalling our similarity definition, we see that at each catalogue entry, global information is taken into account to produce the local score; that is, local scoring accounts for all terms that are contained in the query and all terms/tags that are contained in the content profile and the querying user's and content owner's profiles. Hence, the local score is also a global score.

eXO Top-k Query Analysis. The top-k execution process outlined above is engineered to be very simple and lightweight. It requires only a single phase of communication between the querying node and the catalogue nodes for the query terms. Key role to this plays the way the similarity is defined: on the one hand we want it to take into account both user-user and query-content similarities, as it is necessary in a social network. On the other, each local score of an item to a query is, by construction, also a global score of that item for this query.

Despite valiant efforts, top-k algorithms in decentralized environments can be complex and very expensive in two respects: communication messaging and latency and (disk) IO latency. For instance, KLEE and TPUT [15, 6] require at least two communication phases between the querying node and the nodes holding the index lists and may have to go very deep into the index lists before yielding the results. The key difference in our approach is that we store in each catalogue the whole user and content profiles' info, and this provides the flexibility to compute global score values in every catalogue node. Further, each node only needs to

send the top k entries, yielding a very small bandwidth cost. More formally, we can prove the following.

CLAIM 1. *Consider an object, O, indexed using term set T_O. Given a query Q with query term set T_Q with $|T_Q \cap T_O| = n \geq 2$, the similarity score (for the catalogue entry in each of the n nodes responsible for these TIDs) to the query Q, is exactly the same.*

PROOF. Sketch: Remember that we store the profile of the owner along with every content profile and that each catalogue entry is uniquely defined by its owner's UID (along with a content checksum for content entries). Also remember that the content part of the query is matched against the content part of catalogue entries, while the user part is matched against the accompanying user part. Thus, each of these n nodes would be matching T_Q against the same T_O, hence for any given catalogue entry, the scoring function yields the same score across all n nodes. □

CLAIM 2. *The global list with the top-k most relevant entries stored in the catalogue nodes can be computed in one communication phase, by returning to the querying node the local top-k entries of each visited catalogue node.*

PROOF. Sketch: Assume that an entry which has not been returned to the querying node, belongs to the global top-k. By the definition of our similarity, it follows that this entry must belong in at least one of the local top-k lists returned to the querying node. Then each one of the local top-k lists must have included the specific entry in its reply to the querying node. From 1 we showed that in local lists the scores are global. Hence, it trivially follows that this entry does not belong to the global top k results. □

We employ a boolean weight to compute the cosine similarity. This choice was preferred over a more fine-grained tf-idf weight for two reasons. First, in a social network the majority of content which is being shared by the users is multimedia, whose metadata is usually a short set of descriptive terms without big differences in term frequencies and their body is just binary data. Content profiles created from these content items are low dimensional. This is in contrast to text documents which have analyzable bodies and heavier sets of terms describing their data.

Moreover, to compute a tf-idf weight, global statistics, such as the size of users and the size of content objects in the network, should be computed. In fully decentralized environments this is very complex and expensive and best be avoided. On the other hand, boolean weights as more coarse-grained, do not need global network information. The boolean weight suffers from not taking into account the term frequencies in profiles. This limits the range of score values, so can cause many ties in the scoring process. But the maintenance gain is very more important and we will present later alternative techniques to refine search results.

Content Retrieval Algorithms. After receiving the top-k results, the content retrieval algorithm uses the UID of each catalogue entry to route to the content owners nodes and initiate downloading. As aforementioned, it is unavoidable that different content will be associated with the same score with respect to a query. Think of different pictures of the Acropolis at night, tagged with the terms *Acropolis, Athens, night*. A query with these terms does not really care which of the different photos it receives and all photos with these self-tags will have identical scores to the query. *eXO* exploits this observation to expedite content retrievals, by implementing a new DHT primitive coined *retrieve-m-from-n*.

Specifically, assume a user has requested the top-10 results and *eXO* responded with, say, the top-50 results sharing the top-10 scores. The basic idea of the optimized retrieval algorithms is to, at the first overlay hop, hand over to the next forwarding node a list of the top-50 IDs of content items and their scores. Each node thereafter, looks locally to see to which of the 50 nodes it can route faster. For instance, in Pastry each node can inspect its routing table to see if one of 50 destinations is stored there and hence choose that destination to obtain the desired object. In essence, this optimization provides the flexibility to route to the closest 10 objects among the 50 possibilities.

5. SOCIAL TAG CLOUDS IN EXO

Remember that "social tags" (i.e., terms submitted by other users in order to characterize content or users) are stored on the same node as the original content. More specifically, the set of social tags on some content item is coined the *content tag cloud*. Social tags can also be associated with a specific user, and not just with content items, yielding a *user tag cloud*. The tags constituting a tag cloud can be ranked according to some measure and sorted according to this rank. At this stage, we view this as an orthogonal issue and we leave it to future work. However, for simplicity, within user tag clouds, we sort tags by the number of users having used the specific tag for the same item.

Tagging Process. Social tagging involves two user nodes: the user doing the tagging (*tagger*) and the user owning the tagged object (*taggee*). A tagger issues a tagging request with one or more tags and *eXO* transfers this message to the taggee's node. There, an association of the object and the set of tags is stored, together with the origin user profile of each tag. The association is formed as a content or user tag cloud, indicating the significance of each tag. In addition, on the tagger's side, an inverted list for each tag term is maintained. After a successful tagging operation, the inverted list is updated with the profile of the tagged object. Thus, a bidirectional relation is created between a tagger's and taggee's nodes.

Query Enrichment with Social Tagging. The ultimate goal is to improve query-result quality. Recall the *eXO* approach to indexing content and user profiles: only an owner's tags (a small number) are used to index each object. This is a pragmatic concern in order to avoid overburdening the catalogue nodes with continuous updates to index data, etc. However, it also introduces the potential of missing important items when responding to queries. Consider, for instance, an item with large multi-dimensional semantic content, which requires a fairly large number of terms in order to be adequately characterized, or, an item whose important 'dimensions' change with time. Since only a fraction of these terms may actually be used to index the item within the catalogue nodes, a query specified with these terms will not return the item. For example, an important article involving Obama's high school friends would typically not be

indexed with the keyword "Obama" and therefore a query with keywords "Obama", "youth" will not return the article.

During content (or user-profile) retrieval, *eXO* provides the opportunity to not only download the object, but also to retrieve the social tags (tag cloud) as we visit the content owner's node. Hence, the tag cloud for each retrieved object may be piggybacked onto the response. This way a new set of terms (i.e. those in the tag cloud), submitted by other users and reflecting other opinions, can be discovered by the querying user. Then, a new search query can be formulated, being enriched by tag cloud terms, with the user selecting which tag cloud terms to use based on the terms' relative rank. If the tag cloud is too big, a threshold is used, choosing the most dominant tags of the cloud before piggybacking the cloud onto the response.

Catalogue Updates. Note that the ranking of tags within a cloud is performed at the owner's node, reflecting the owner's perception of the importance of each cloud term for his object. When ranks of these terms exceed some user-perceived importance threshold, our design permits the owner to update the "owner tags"; that is, utilize high-ranked social tags, and index the object using these tags as well. This mechanism is employed in order to cope with time dynamics, especially for object's whose important "dimensions" change with time. That is, an owner can decide, based on the tagging activity on an object, that new terms can be used to index the object and make it visible and appropriate from now on.

6. PERSONAL SOCIAL NETWORKS

Up to this point, we have elaborated how the DHT-based catalogue served as the basis for discovering interesting items when responding to a query and for retrieving these items. Social networks, however, are ad hoc in their essence, and basing all functionality upon the DHT may seem too stringent and artificial.

Building Personal Social Networks. To this extent, *eXO* harnesses the extra information supplied by the user tagging activity to further advance the state of the art.

As mentioned earlier, a bidirectional relation between the tagger's and taggee's nodes is created with every tag; this also holds for tags on user profiles. These couplings exist in the form of cross-referenced UIDs stored locally on the tagging/tagged nodes and not as separate network connections or routing table entries. Further, note that users can search for users similar to their profiles, or for experts in specific domains, using relevant terms, and so on; then, through the tag clouds returned by such queries, the users can further discover new relevant tags and new related users, resulting in the emergence of and association with specific social groups (be it of social friends, professional acquaintances, experts in domains of interest, etc).

These virtual unstructured networks can then materialize through "connection requests": the local node sends a befriending request to the remote nodes; if such a request is accepted by the remote user, the two communicating nodes keep each other's UID in separate "acquaintance" lists. *eXO* users may then opt to share part of their information with only members of these social groups, or direct queries towards specific groups that are expected to return more and/or higher quality results.

Querying Personal Social Networks. Putting it all together, queries in *eXO* can be of three types: (i) queries directed to the DHT catalogue nodes only; (ii) queries disseminated through the personal social networks; and (iii) queries directed in parallel to both of the above mechanisms.

In the first case (see section 4), queries are directed to the DHT catalogue nodes using which the top-k items for the query are identified, based on catalogue information (owner tags). Additionally, querying users can retrieve any tag clouds associated with the items and reformulate the query using important social tags. This process can be repeated until the user is satisfied with the results.

In the second case, queries are directed to the querying user's appropriate social network(s). If no appropriate social network exists, the user can establish it first as outlined above and then issue the query. The search query visits each of the users that are neighbors in the social net. At each neighbor, the local tagger's data structure is searched locally for entries which, like catalogue entries, contain user/content profiles and UIDs, and they are returned to the querying user. It is likely that these inverted lists are small, although a similarity processing like the one we mentioned in the previous subsections, can be applied to return the most relevant to the query results. Thus, these results can constitute a complementary list of search results, which have been obtained by *eXO*'s similarity query engine, retrieval algorithms, and tagging metadata. Querying users can decide how deep into the query-relevant social network they should delve before they stop.

Finally, in the last case queries are directed in parallel to both the DHT catalogue nodes and to the relevant unstructured social network. Thus, a new query can proceed in parallel to the catalogue nodes and to the nodes making up the related user-group.

Link prediction. The system, currently, does not perform link prediction per se. Instead, it is user-centric. Each user can utilize the public network's infrastructure to identify other social links of interest by issuing specific user-profile queries. Retrieving these profiles and their tags, they can reissue more elaborate queries and thus establish/enrich their PSNs. Having done this, more elaborate algorithms can be employed to perform link prediction, for example by utilizing *gateway* nodes, belonging to more than one PSN, through which to route befriending requests. Alternatively, as PSN nodes share elaborate, private profiles of their friends, they can detect common friends, which are not themselves direct friends, and advise them to link-up. This is a straightforward activity facilitated and easily supported by *eXO*.

7. EXPERIMENTATION

We implemented *eXO* over FreePastry[2]. We ran our experiments on an 8-core, 30GB RAM system. Due to space reasons, we report on only type 1 queries.

Methodology. We used two real datasets from Flickr and Facebook. The Flickr dataset was formed from the Tagora Project[3], containing associations among users, resources (e.g.

[2] `http://freepastry.org/FreePastry/`
[3] `http://www.tagora-project.eu/data/`

photos), and tags. With this we created content profiles for shared items. We used a slice of around 49,832 content items corresponding to 1000 users and to 264,379 terms. Second, we systematically crawled Facebook and formed a set of user profiles from public user data. Public profiles contain a sample of users' friends (at most 8 in our data) and a collection of users' interests and tastes, given as hyperlinks and phrases. We extracted this information (36,261 user profiles) and placed it in XML files which fed our tests. Before each experiment starts, every network node was initialized and allowed to exchange necessary messages to construct its routing table. Then, the indexing phase begun. The indexing concerned both user and content data. When the indexing process was completed, the system was ready to execute the test scenarios. We then composed queries using real user and content profiles. The first kind of queries were formed by randomly selecting a set of terms from the indexed Flickr content items, whereas the second one by choosing one by one the terms from random user profiles indexed in the network. We varied the number of query terms, the value of k in each query, and the network size.

Performance Metrics. We measured the average number of messages (hops) on the network for each search or index request in relation to the network size. Moreover, the average number of bytes (bandwidth) for each query result list was computed. Bytes are counted for all the result list data returned from the catalogue nodes to the query source node. Last, we measured the node load, computed as the number of times each node is visited during query processing. The load was compared to the document frequency distribution of each indexed term to appropriately explain the results. "Document" frequency in our approach refers to the number of content or user profiles that contain a specific term. Please note that randomly selecting query terms from the set of all terms does *not* result in uniform-random loads, since different terms have substantially different document frequencies. Also note that the load balancing issues are largely orthogonal to this work: any off-the-shelf approach can be utilized to deal with extremely popular terms, for example, that could overburden an index node responsible for this term. Note that such issues are inherent into any design, be it DHT-based or not, since nodes with popular content will be hit more frequently than others.

Results. Figure 2(a), depicts the load distribution among nodes for the Flickr dataset, showing a balance in load, except for some relatively mild peaks. The peaks are justified if we take a look at the Flickr dataset's "document" frequency distribution graph in figure 2(b). The "document" frequency, for the Flickr dataset contains a number of terms which are dominant, explaining the peaks. Please note that, in any case, any load distribution strategy can be employed in cases where load balancing issues emerge. Figure 3(a) presents the load distribution in network nodes for the Facebook dataset, while 3(b) show the relevant "document" frequency. We observe a fairly balanced distribution of node hits. Also, the percentage of the total hits that any node receives, is very low. Thus, we expect no bottlenecks at catalogue nodes, despite the non-uniform distribution of the terms' "document" frequency.

The next experiment involves the indexing of 8,000 user profiles (one per node), and 160,000 queries, for various

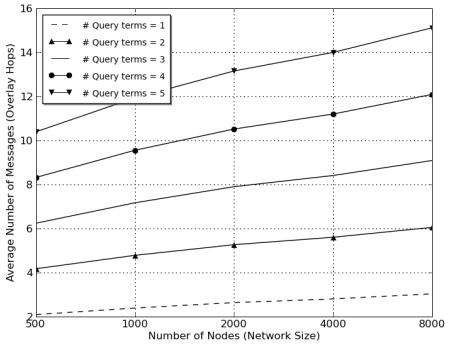


Figure 4: Number of messages versus network size

Table 1: Average Bytes Per Query Result

Top-K	# Query Terms	Bytes
Flickr Dataset		
10	2	2.2k
20	2	4.3k
50	2	10.8k
100	2	21.8k
Facebook Dataset		
10	1	3.7k
10	2	7.5k
10	3	11.2k
10	4	15.5k
10	5	19.1k

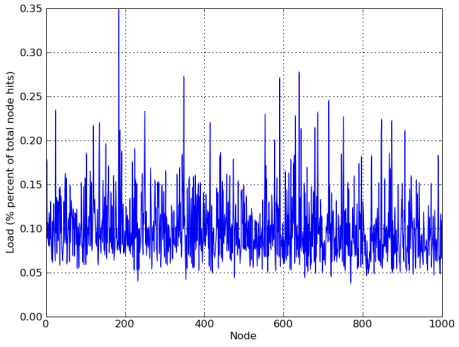
network sizes. We computed the average number of overlay messages to the destination catalogue nodes for various query sizes. Figure 4 shows results for queries with one to five different keywords. The results confirm the logarithmic relation between network size and number of messages for each search request. As mentioned earlier, a search request initiates a number of parallel routing requests, one request for every keyword in the query. So we expect to have multiples of the first curve in our graph depending on the number of query keywords, which we indeed observe.

Finally, table 1 presents the number of bytes communicated between the source and the catalogue nodes. For this experiment, we issued 20,000 queries per test and computed the average payload for those queries which returned exactly the maximum total number of result lists (a multiple of K). The results show the average size of communicated data per query to be very low.

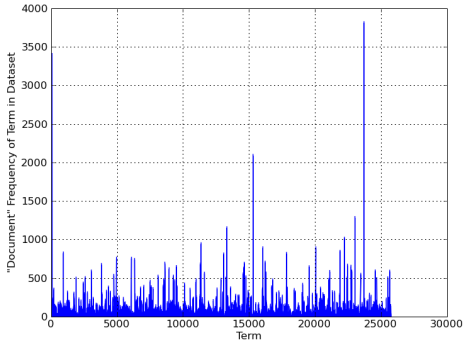
Overall, the results confirm that: (i) scalability is ensured with respect to a growing network, given (a) the relation between the number of messages and the network size, (b) the fairly well-balanced load among the network nodes, and (c) the small size of bandwidth consumption for all queries; (ii) the parallel nature of the search-query process ensures that the user-perceived latency will be low, as the results are computed in just one communication phase.

8. RELATED WORK

The backbone of existing approaches for top-k query processing is the classic paper of Fagin, et al.[10], which has been extended to distributed environments in works such as

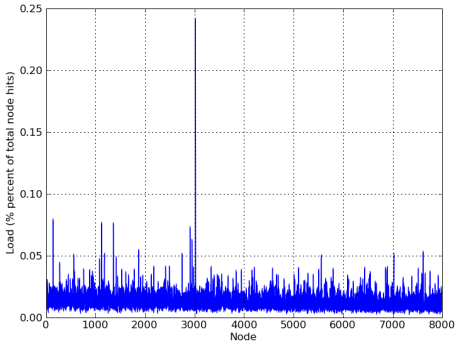


(a) Load Distribution

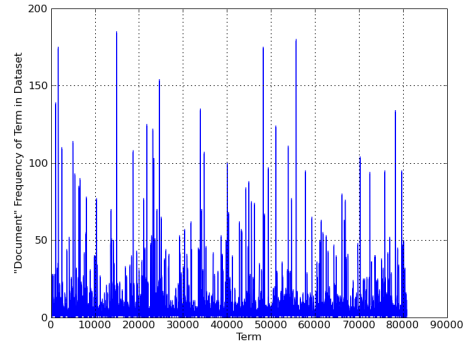


(b) Term "Document" Frequency Distribution

Figure 2: Flickr Dataset



(a) Load Distribution



(b) Term "Document" Frequency Distribution

Figure 3: Facebook dataset

KLEE[15] and TPUT[6]. However, all these approaches are quite expensive. KLEE and TPUT contribute an overhead of at least two phases of communication among the query source and destination nodes. In comparison, our approach requires only a single phase of communication.

Social web search has also been studied in [16], where web searching results are appropriately enhanced by the social links' data. Top-k processing in collaborative tagging sites has been studied in [23] where user networks are created (e.g. based on the number of same tags they used for tagged items, or on the overlap of their tagged items) and uses generalized versions of Fagin's algorithms to exploit social tags. An item is deemed relevant to a query issued by a user u based on how many users in u's network have tagged this item with a query term. [5] presents top-k algorithms for selecting the most relevant tags given a document. *eXO* provides mechanisms to support the building and utilization of such networks.

In [1] a gossip protocol, is employed so to associate users with similar tagging behavior and to develop enough state locally for top-k queries to execute swiftly. This is in contrast to our *eXO* design where a DHT is used for this purpose. Note that using a DHT and our top-k approach will permit faster retrievals and associations with other users, but lacks the flexibility afforded by gossiping over a completely unstructured network.

In [19] top-k processing employs similarity functions taking into account features such as friends and (friends of friends), and fine-grained relations of tags to documents and profiles and tag expansions are defined. In *eXO* we also provide similarity definitions that consist of query- and user-specific parts and can be enhanced by social tags. Although not allowing for transitive friendships yet, this can be implemented too by visiting direct friends and issuing queries for their friends etc. However, this will be much costlier in a decentralized environment than the centralized environment in [19]. Finally, the top-k algorithm in [19] considers additional information that is not easily available in decentralized settings, such as all the documents tagged by any user, all the tags used by any user, etc.

Related work in P2P searching includes the Minerva and Galanx [3, 22] DHT platforms. Like *eXO* Minerva maintains indices on the DHT. However, these indices index different information than our catalogues, query processing involves a multiphase top-k algorithm to return the appropriate results, no special content retrieval algorithms exist, and there is no support for social tagging, social query expansion, building personal networks, and relevant scoring functions. High costs also occur in Tagster[12]: based on a vector space model, it adds extra overhead for the computation and maintenance of global statistics, such as the document frequencies. Global statistics are costly to com-

pute and maintain despite efforts such as [2].

With the exception of [1], none of the above addresses the issues of decentralized scalable and efficient social networking. The gossiping protocols presented in [1] could be combined with *eXO* to incorporate the benefits of gossiping-style discovery and search. This is left for future work.

9. LIMITATIONS AND OPEN ISSUES

This work was a first crack at designing and implementing fully decentralized and widely distributed social networking services, bringing the power and burden of content ownership and sharing to the end users at the edge of the network. Our system provides efficient decentralized support for the cornerstones of social networks: friend lists, user/content tagging, and distributed processing of keyword/profile-aware queries. However, several issues remain open for future work.

Privacy and Availability. First and foremost, our work does not deal with content availability for privately/intimately shared content. One could devise and advocate some secret-sharing technique to allow private content to be replicated across the network so as to boost its availability without compromising its privacy. Any candidate scheme should be flexible enough to allow for frequent membership changes, while imposing a low network overhead; a formidable task in its own. Replication to "friends" (who already have access to private content) would be another attractive route; this solution would in turn call for a technique to transparently and remotely revoke access rights and replicated content from "x-friends". Moreover, even public data replication is an open issue in its own, if one further desires to perform it so as to enhance query performance in addition to data availability.

Meta-services. Second, social network users are accustomed to a specific "workflow" and meta-services, such as news feeds and games. The former could easily be implemented via an elementary publish/subscribe system of sorts; it would suffice to send notifications of new events to all nodes in the friend list of a given node. More elaborate schemes may also be devised (e.g., using dissemination trees or application-level multicast techniques). Similar solutions could also be applied to distributed games, albeit the requirements with regard to the latency of notification propagation will be much more stringent. All this may stress the network overlay, calling for new (possibly hierarchical) structures, such as those outlined in [17], or even completely novel overlays, specifically tailored for the needs of decentralized social networking.

Gossiping and PSNs. Third applying gossip- or broadcast-based protocols to the design of our personal social networks may yield some very interesting results. Directed broadcast has already been studied in DHTs[7]; further examination and application of such solutions in our environment remains an open issue. Similarly, gossiping algorithms may be employed to traverse the social graph (viewed as the union of all PSNs) and identify new social links, as is advocated in [4]. This is an interesting issue. One must decide on which metrics and features to use in order to establish further social ties between users. For example, identifying common neighbors, aggregating similarity weights across several traversed PSN links, deciding on the appropriate summary structures to use to represent such features (which are propagated through PSNs) are open, exciting issues.

Anonymity, Autonomy, and Privacy. These are conflicting goals. Specifically, owners may wish to know the identity of requesting users, before they grant a request to access local content. Striking a balance between the goals of autonomy and anonymity is an open issue. However, note that the decentralized nature of *eXO* does not create as serious of an anonymity problem as that of centralized systems. In the worst case, querying users may have to reveal their identities to the users whose content they wish to retrieve. However, there is no globally available record and log which records a users' accesses, behaviors, and social ties. Viewed differently, to uncover private data (e.g., the user's PSNs, the content items accessed by her, etc.) a large distributed set of meta-data must be accessed, which renders the task more difficult. Even more importantly, there is no single, central organization which maintains this information and can use it serendipitously or maliciously.

Distributed collaborative filtering. *eXO* can be leveraged to perform collaborative filtering. For instance, to recommend (perhaps using continuous background queries) new items of interest to users, based on what other users/friends are doing. In this direction an infrastructure that can perform the following tasks would be highly desirable:

- Monitor events of interest (such as when a user expresses an interest on an item, or highly tags/evaluates an item, etc).
- Collect such behavior information efficiently from the public network and PSNs.
- Detect when new recommendations can be effectively made to others (minimum support levels for similarity and number of event occurrences, etc).
- Make recommendations to those appropriate friends, based on friends profiles.
- Figuring out what is new to some friends and what they already know about.

Doing all this in a decentralized manner is a challenging issue.

Last, there are several information retrieval related issues that our work has not touched. For example, selection of the best tags to use for indexing or of the most promising tags in a tag cloud to use for query enrichment, stopword-like filtering of highly popular tags, and distributed novelty-based computations on the set of events delivered to recipients of news feeds, are just a few of them.

10. CONCLUSION

We presented *eXO*, a novel decentralized social network. *eXO* consists of (i) mechanisms for indexing content and related metadata, (ii) appropriate similarity functions with social networking and traditional query-to-content relevance, (iii) efficient top-k algorithms for search-for-user and search-for-content queries, (iv) efficient overlay algorithms for retrieving content, exploiting the architecture of the overlay network on the one hand, and the characteristics of content in social network sharing environments on the other, (v) scalable social tagging mechanisms and methods which

exploit social tags to improve the quality of query results, (vi) methods for building social networks, (vii) discovery and retrieval of related content via a combination of DHT-based indexing and of unstructured, query-relevant, social networks and (viii) an implementation of the system and a performance evaluation substantiating the claims for scalability and efficiency. With this work we have endeavored to showcase the appropriateness and feasibility of fully decentralized social networking services. The widespread adoption of Diaspora has testified that the users are interested in and probably ready for such a move. We therefore put forth this paradigm as our route of choice.

11. REFERENCES

[1] X. Bai, M. Bertier, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. Gossiping personalized queries. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, pages 87–98, 2010.

[2] M. Bender, S. Michel, P. Triantafillou, and G. Weikum. Global document frequency estimation in peer-to-peer web search. In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, 2006.

[3] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Minerva: Collaborative p2p search. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2005.

[4] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. The Gossple anonymous social network. In *Proc. IFIP/ACM IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware)*, 2010.

[5] A. Budura, S. Michel, P. Cudré-Mauroux, and K. Aberer. To tag or not to tag – harvesting adjacent metadata in large-scale tagging systems. In *Proc. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR)*, 2008.

[6] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proc. ACM Symposium on Principles on Distributed Computing (PODC)*, pages 206–215, 2004.

[7] M. Castro, M. Costa, and A. Rowstron. Should we build gnutella on a structured overlay? In *Proc. HotNets II*, 2003.

[8] P. A. Dmitriev, N. Eiron, M. Fontoura, and E. Shekita. Using annotations in enterprise search. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 811–817, New York, NY, USA, 2006. ACM.

[9] P. Druschel and A. Rowstron. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware)*, 2001.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001.

[11] S. A. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 2006.

[12] O. Görlitz, S. Sizov, and S. Staab. Tagster - tagging-based distributed content sharing. In *Proc. ESWC*, 2008.

[13] P. Heymann, G. Koutrika, and H. Garcia-Molina. Can social bookmarking improve web search? In *Proc. Intl. Conf. on Web Search and Web Data Mining (WSDM)*, pages 195–206, New York, NY, USA, 2008. ACM.

[14] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proc. Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[15] S. Michel, P. Triantafillou, and G. Weikum. KLEE: A framework for distributed top-k query algorithms. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 637–648, 2005.

[16] A. Mislove, K. P. Gummadi, and P. Druschel. Exploiting social networks for internet search. In *Proc. HotNets*, 2006.

[17] N. Ntarmos and P. Triantafillou. AESOP: Altruism-endowed self-organizing peers. In *Proc. Intl. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2004.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2001.

[19] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *Proc. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 523–530, New York, NY, USA, 2008. ACM.

[20] R. Schenkel, T. Crecelius, M. Kacimi, T. Neumann, J. Xavier Parreira, M. Spaniol, and G. Weikum. Social wisdom for search and recommendation. *IEEE Data Engineering Bulletin*, 31(2):40–49, June 2008.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 149–160, 2001.

[22] Y. Wang, L. Galanis, and D. J. de Witt. GALANX: An efficient peer-to-peer search engine system. `http://www.cs.wisc.edu/~yuanwang`.

[23] S. A. Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 1(1):710–721, 2008.

[24] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U. of California at Berkeley, CSD, 2001.

Database Access Control & Privacy: Is There A Common Ground?

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

Data privacy issues are increasingly becoming important for many applications. Traditionally, research in the database community in the area of data security can be broadly classified into access control research and data privacy research. Surprisingly, there is little overlap between these two areas. In this paper, we open up a discussion that asks if there is a suitable middle-ground between these areas. Given that the only infrastructure provided by database systems where much sensitive data resides is access control, we ask the question how the database systems infrastructure can step up to assist with privacy needs.

1. INTRODUCTION

Data privacy issues are becoming increasingly important for our society. This is evidenced by the fact that the responsible management of sensitive data is explicitly being mandated through laws such as the Sarbanes-Oaxley Act and the Health Insurance Portability and Accountability Act (HIPAA). Accordingly, data privacy has received substantial attention in previous work [3]. The key technical challenge is to balance utility with the need to preserve privacy of individual data. Initial work on data privacy focused on data publishing where an "anonymized" data set is released to the public for analysis. However, the evidence increasingly points out the privacy risks inherent in this approach [9]. It is now believed that a query-based approach where the database system *answers a query in a privacy-sensitive manner* is generally superior from the privacy perspective to the data publishing paradigm [9].

However, the only support provided by database systems where much sensitive structured data reside is the mechanism for *access control*. Briefly, the idea is to authorize a user to access only a subset of the data. The authorization is enforced by explicitly rewriting queries to limit access to the authorized subset. Such a model of authorization is intuitive to application developers and users of the database system. The programming model remains the same as before — (1) data is accessed using SQL queries, (2) the results returned are deterministic, hence the utility of query results is clear, (3) there is no restriction on the class of queries that can be executed, and (4) there is no restriction on the total number of query executions. In other words, an access control mechanism is fully compatible with the functionality of a general purpose database system. Not surprisingly, access control is supported in all commercial database systems and the SQL standard. (In fact, some commercial database systems also support fine-grained access control [11].)

The main limitation of the traditional access control mechanism in supporting data privacy is that it is "black and white". Consider for example advanced data analysis tasks such as location-aware services of operational Business Intelligence that need to stitch together multiple sources of data such as sales history, demographics and location sensors many of which have sensitive data. For these examples, effective analytics needs to leverage many "signals" derived by aggregating data from sources who have sensitive private information, at the same time without revealing any sensitive individual information. But the access control mechanism offers only two choices — (1) release no aggregate information thereby preserving privacy at the expense of utility, or (2) release accurate aggregates thus risking privacy breaches for utility.

There is considerable previous work in privacy-preserving query answering that goes beyond the "black and white" world of access control [1, 8]. The class of queries is generally restricted to aggregate queries and the approach adopted broadly is to add noise to the aggregates. However, non-aggregate queries are a large class of database queries and the support for them is rather limited in the above bodies of previous work.

In this paper, we ask if we can get the best of both worlds — combine the advantages offered by access control mechanisms while at the same time going beyond the "black and white" world by leveraging previous work on privacy preserving query answering. We can break down this question as follows: (1) What is the database API that combines traditional access control mechanisms with privacy-preserving query answering? (2) How do we implement the suggested APIs in a principled manner? (3) How do we mix and match both mechanisms to ensure privacy guarantees?

We explore a natural hybrid system that combines (a) a set of authorization predicates restricting access per user to only a subset of the data, and (b) a set of "noisy" views that (as the term suggests) expose perturbed aggregate information over data not accessible through the authorization pred-

5th Biennial Conference on Innovative Data Systems Research (CIDR '11)
January 9-12, 2011, Asilomar, California, USA.

icates. For example in an employee-department database, we could allow employees to see their own employee record and also publish a "noisy" view exposing the average salary of the organization. This hybrid potentially has significant advantages. Accessing data through a set of views is natural for users of database systems and thus provides a simple extension to today's database API. It offers the functionality of access control for queries that refer only to the database tables and views, and the privacy guarantees of previous work for queries that only refer to the "noisy" views. In addition, we obtain value beyond the sum of the individual parts by allowing rich queries that refer to *both* database tables and "noisy" views.

We implement the "noisy" views by using previous work on implementing *differential privacy* [5, 8]. In order to answer question (3) above, we introduce the notion of differential privacy *relative to an authorization policy* and explain its desirable theoretical properties. We show that the seemingly ad hoc hybrid system described above satisfies differential privacy relative to the authorization policy.

We note that it is possible to expose the APIs we propose by extending libraries supporting differential privacy such as PINQ with access control mechanisms. However, given that the recent trend in the industry is to implement data security primitives in the database server, our discussion in this paper is presented as a modification to the database server. We think of our paper more as a first step in initiating discussions on how database systems can provide meaningful support to address privacy concerns. We comment on open issues, including a critique of state of the art privacy models.

2. REVIEWING ACCESS CONTROL

There has been a lot of work in the area of access control in databases. The idea with access control is that each database user gets access to a subset of the database that the user can query. The current SQL standard allows coarse grained access both to database tables as well as views.

Recent work [2, 6, 11, 13] has emphasized supporting predicate based *fine-grained* access control policies in the database server. For example, we wish to be able to grant each employee in an organization access their own record in the employee table.

In this paper, we consider fine-grained access control policies. We assume that access control policies expose per user a subset of each database table (this is the approach adopted by some commercial systems like Oracle VPD). The policy is formally captured by specifying for each user and each database table, an *authorization predicate*. Since the number of users can be potentially large, the predicate is specified succinctly as a parameterized predicate that references the function *userID*() that provides the identity of the current user. For example, we can grant each employee access to their own record as follows. (We adopt the syntax proposed in previous work [2].)

```
grant select on employee
      where (empid = userID())
to public
```

Access can be granted not only to tables but also to views. In this way, we can expose additional information such as aggregate information. For example, we can create a view that counts the total number of employees and grant access to the view to every employee. For ease of exposition and

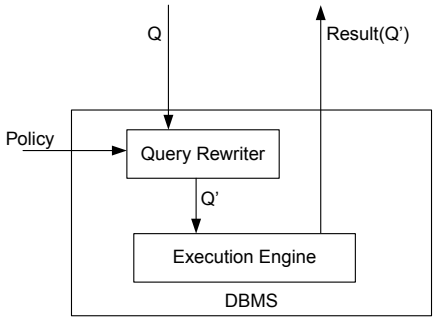


Figure 1: Leveraging Fine-Grained Access Control

without loss of generality, in the rest of this paper, we restrict authorization predicates to only be specified for tables.

We now formalize the notion of an *access control policy*.

Definition 1. A *access control policy* $\mathcal{P}$ specifies for each user and for each database table, a corresponding authorization predicate. We also refer to an access control policy as an *authorization* policy.

We assume that the function *auth(T, u)* denotes the authorization predicate on table T corresponding to user u. For instance, in the above example *auth(T, u)* is the predicate `empid = u`.

Let the tables in the database be $T_1, \ldots, T_k$. Fix a database instance A. We refer to the vector $<\sigma_{auth(T_1,u)}(T_1), \ldots, \sigma_{auth(T_k,u)}(T_k)>$ as evaluated on the instance A as the *authorized subset* for user u, denoted $\mathcal{P}(u, A)$. (In all our notation, if the user is clear from the context, we drop the reference to the user.)

Queries are executed by *rewriting* them to add the authorization predicates. For example, the query:

```
select salary from employee
```

gets rewritten to:

```
select salary from employee
where empid = userID()
```

In the same way, update statements are also rewritten to go only against the authorized subset. For example, the update:

```
update employee set nickname = 'Jeff'
```

is rewritten as follows:

```
update employee set nickname = 'Jeff'
where empid = userID()
```

We note that in general, the access control policy can allow a different "pre-update" authorization predicate than the one for queries. Further, previously proposed access control policy languages also support the notion of a "post-update" authorization predicate which checks whether the new values of the updated rows are authorized. For ease of exposition, we assume that (1) there is only one authorization predicate used for both queries and pre-update and that (2) there are no post-update predicates, while noting that our techniques and results extend if we relax this assumption.

Figure 1 illustrates the infrastructure supporting access control mechanisms in a database system.

2.1 Limitations

As we discussed in Section 1, granting access to accurate aggregations over different subsets of the data can potentially leak information. For instance, in an Employee-Department database, suppose we grant a data analyst access to the number of employees at various levels of the organizational hierarchy grouped by the department, gender and ethnicity. If the analyst knows an employee with a rare ethnicity, the level of the employee could potentially get breached. Basically, the choices offered by access control mechanisms are "black-and-white"; we can grant access either to accurate aggregate information thereby compromising privacy, or to no aggregate information compromising utility.

Previous work in data privacy has studied techniques for releasing information while preserving privacy allowing us a middle ground in the above scenario. Briefly, the idea is to add noise to the result of a computation. We next address the question of how the database API can be extended to exploit this previous work.

3. NOISY VIEWS

In Section 1, we proposed the notion of *noisy views* as a possible abstraction for integrating privacy mechanisms in a traditional database. The core properties of noisy views that we support may be summarized as follows:

- *Noisy Views are a DDL construct:* Noisy views may be defined by a data provider in the same way a traditional view is declared, e.g., through a CREATE VIEW statement.
- *Noisy Views are Non-Deterministic:* Although the DDL expression for a noisy view is no different from that of a traditional view, their semantics is non-deterministic. Intuitively, they correspond to the result of executing the DDL associated with the noisy views but enriched with a random "noise" to ensure no privacy leaks happen.
- *Noisy views and Access Control co-exist:* A query can reference both a noisy view and other authorized objects.
- *Noisy Views are Access Control Aware:* Noise is only added to the part of the result derived from unauthorized data. The part derived from authorized data is not perturbed.
- *Noisy Views are Named:* To reflect to the application developer the fact that noisy views are non-deterministic, the noisy views need to be explicitly named, much like views in the SQL standard.
- *Traditional DBMS Execution Engine:* The DBMS query execution engine is left unchanged. Only minimal changes to the database system are needed: (1) a noise-injecting function and (2) a modified query rewriter that rewrites a reference to a noisy view.
- *No Change in Data:* The privacy-sensitive database content is left unchanged.

The above properties summarize the core facets of a noisy view object. The advantages of this approach are that (1) it builds on the familiar notion of views thereby inheriting the benefits of access control mechanisms, and (2) by requiring that queries access the noisy views by explicitly naming them, it clearly separates the deterministic components from the non-deterministic components of the system. Any query that does not name the noisy views will have the same behavior as with just access control mechanisms.

However, the specific details of the privacy model being supported determines several additional details: (1) What subset of SQL can be supported as noisy views? (2) How exactly is the noise added? (3) What is the additional information that needs to be specified: for users as well as queries? (4) What are the privacy guarantees? The answers to the above questions critically depend on the specific privacy model that is adopted.

The rest of the paper answers precisely these questions for the well-known model of differential privacy [5]. We first review state of the art differential privacy in Section 4 and then describe our implementation in Section 5. Finally, we note that the model of noisy views is more general and provides a template for capturing privacy models in database systems like the approach adopted by Netz et al. [10] for encapsulating data mining models in databases.

4. REVIEWING DIFFERENTIAL PRIVACY

We now briefly discuss *differential privacy* [5] which is considered to be the current state of the art in privacy models.

4.1 Definition

Intuitively, differential privacy requires that computations be formally indistinguishable when run with and without any single record. The following definition makes the intuition precise. We denote the symmetric difference between two data sets A and B as $A \oplus B$ (for a database with multiple tables, we take the union of the symmetric difference of the corresponding tables).

Definition 2. A randomized computation M provides ϵ-differential privacy if for any two database instances A and B and any set of possible outputs $S \subseteq \mathit{Range}(M)$:

$$\mathbf{Pr}[M(A) \in S] \leq \mathbf{Pr}[M(B) \in S] \times \exp(\epsilon \times |A \oplus B|)$$

□

The parameter ϵ quantifies the degree of privacy. A smaller value of ϵ corresponds to a stronger guarantee — for example if we set ϵ to be 0, then M is constrained to produce the same output independent of input. On the other hand, a larger value of ϵ indicates a weaker guarantee.

Intuitively, differential privacy ensures that adding an individual record to the database does not reveal much additional information. Thus, an adversary would not be able to learn if any particular data item was used as a result of this computation. Perhaps the biggest advantage of differential privacy is that it makes no reference to (and hence no assumptions about) background knowledge. It thus relieves us from the burden of changing privacy models as assumptions about background knowledge change.

4.2 Implementation Using PINQ

Most techniques implementing differential privacy such as Privacy Integrated Queries (PINQ) [8] focus on aggregations (our discussion below is also based on PINQ). Aggregates are supported by *output* perturbation — the original aggregate is first computed and then perturbed by adding random noise (thus the noise corresponds to an absolute error in the output).

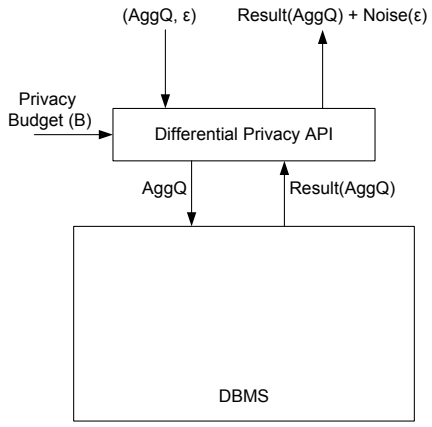


Figure 2: Leveraging Differential Privacy

Figure 2 illustrates how an application can leverage PINQ. Each aggregate query *AggQ* is issued with a privacy parameter ϵ (see Definition 2). The query execution algorithm guarantees ϵ-differential privacy by adding a carefully chosen random noise to the output; the noise is chosen as a function of ϵ and the aggregation being performed. The noise added is inversely related to ϵ — a larger value of ϵ (weaker privacy guarantee) can be accommodated with a smaller noise, whereas a smaller value of ϵ (stronger privacy guarantee) requires more noise.

As more queries are run, the overall privacy guarantee gets weaker. Formally, we have the following previously published result [8].

THEOREM 1. *Let M_i each provide ϵ_i-differential privacy. Then the sequence of M_i provides $\Sigma_i \epsilon_i$-differential privacy.*

Thus, overall the system satisfies $\Sigma_i \epsilon_i$-differential privacy where the i^{th} query is run with parameter ϵ_i.

There is previous work [4] that formally shows that if an unbounded number of queries are allowed, then eventually privacy is breached. Therefore, the notion of a *privacy budget B* is introduced that bounds the number of queries a user can run. As each query is run with its privacy parameter ϵ, the budget is decremented by ϵ. Queries can only be run so long as permitted by the remaining budget. A larger privacy budget allows a larger number of queries to be run but with a greater risk of privacy breach. Thus, both the budget and the query-specific privacy parameters can be used to trade off privacy with utility.

PINQ supports differentially private variants for all the standard SQL aggregations such as `sum`, `count` and `average`. We refer to the differentially private variants respectively as `noisySum`, `noisyCount` and `noisyAvg`. The aggregations can be computed over a restricted class of SQL operations such as filters and key-key joins. By using a partitioning operation, it also supports a limited form of grouping. Since the implementation only adds noise to the output of queries, all of the query processing can be done using the DBMS execution engine without modifying the underlying data (this is in contrast with input perturbation techniques [3]).

4.2.1 Limitations

As noted above, using differential privacy requires the programmer to set various parameters and also understand that an unbounded number of queries cannot be run by the same user. While the meaning of the parameters is intuitive, choosing appropriate values for them is non-trivial. The random noise added is a function not only of the privacy parameter ϵ but also the *sensitivity* of the aggregation, which is the maximum influence any single record can have on the output of the aggregation. Differential privacy implementations work best for low-sensitivity computations. For example, the sensitivity of aggregates such as `count` is low. On the other hand, for aggregates such as `sum` the sensitivity can be arbitrarily large (that said, when a large number of records are being summed, a large absolute error can be tolerated since it might not correspond to large relative error). For a given privacy guarantee ϵ, we need to add significantly more noise as the sensitivity increases. On the other hand, answering higher sensitivity queries without increasing the noise reduces the total number of queries that can be executed within the privacy budget. Such interactions between the parameters makes them difficult to set. We note that the above limitations hold not only for PINQ but for all previously proposed differential privacy algorithms.

Although differential privacy comes with the above "baggage", it offers a principled mechanism to navigate the privacy/utility trade off. This is in stark contrast to the "black-and-white" world of access control. Further, recent empirical work has begun to apply differential privacy to several real-world data analysis tasks successfully [8, 14]. Given this fact, it is natural to ask if we can implement our noisy view abstraction through differential privacy primitives. We study this question next.

5. DIFFERENTIALLY PRIVATE NOISY VIEWS

In this section, we discuss how we implement noisy views based on differential privacy and discuss how it can be integrated in a database system. We term noisy views implemented using differential privacy as *differentially private views* or *DPViews* in short. We present our system as an enhancement of the database server while noting that much of the functionality can also be supported through middleware requiring no changes to the server, say by enhancing PINQ with access control primitives.

Since the class of DPViews we consider is influenced by our privacy guarantees, we begin this section by discussing the privacy guarantee we seek to provide.

5.1 Differential Privacy Relative To Views

The main challenge in formalizing the privacy guarantee is that we do not want to charge the system with protecting the privacy of information that is revealed through the authorization policy. We illustrate with an example. We assume that the authorization predicates are known to all users.

Example 1. Suppose that in an organization, a user is authorized to see the records of all employees whose salary is greater than \$100000. Even though the user is not authorized to see the records of other employees, he/she knows that their salary is less than or equal to \$100000. □

A user knows that the underlying database has to be consistent with the authorized subset. This is how information

is revealed about the overall data. Accordingly, we introduce the notion of differential privacy *relative to* an authorization policy as follows.

Definition 3. We say a randomized computation M provides ϵ-differential privacy *relative to an authorization policy* $\mathcal{P}$ for user u if for any two database instances A and B such that $\mathcal{P}(u, A) = \mathcal{P}(u, B)$ and any set of possible outputs $S \subseteq Range(M)$:

$$\mathbf{Pr}[M(A) \in S] \leq \mathbf{Pr}[M(B) \in S] \times \exp(\epsilon \times |A \oplus B|)$$

□

We note that the main difference from the usual notion of differential privacy is that we only consider instance pairs that agree on the authorized subsets. Definition 3 offers us a principled way to reason about privacy in the context of a given access control policy. At one end, if the user is not granted access to any data, then Definition 3 reduces to standard differential privacy. By granting access to more data, the system is only charged with providing a weaker privacy guarantee.

Finally, we carry out the discussion in this section for a single user noting that all results generalize to multiple users. So the references to the user are dropped in the notation.

5.2 Overall Architecture

The overall policy specified to the system initially consists of an authorization policy and a set of DPViews. We use the access control infrastructure to grant and/or deny access to the DPViews in the same way as with traditional views.

In general, a query can reference database tables and DPViews. An update can only reference database tables. Queries and updates are executed by rewriting them in a way that guarantees differential privacy relative to the authorization policy $\mathcal{P}$.

References to database tables are rewritten as described in Section 2 to reflect the authorization policy. For queries and updates that do not reference DPViews, the behavior is identical to only having an authorization policy. The differential privacy guarantee is obtained via the following result.

THEOREM 2. *Fix an authorization policy $\mathcal{P}$ and a query that does not refer to DPViews. The rewritten query is 0-differentially private relative to $\mathcal{P}$. Similarly, an update (that is not allowed to refer to DPViews) when rewritten is 0-differentially private relative to $\mathcal{P}$.*

PROOF. We can think of the authorization semantics for queries and updates as follows. The database instance A is replaced with the authorized subset $\mathcal{P}(A)$ and the original unrewritten statement (query or update) is run on this smaller instance. The result follows. □

Any reference to a DPView in a query is made with a privacy parameter ϵ (the ϵ here refers to Definition 3) and rewritten in a way that guarantees ϵ-differential privacy relative to the authorization policy. The way in which DPView references are rewritten is described in Section 5.3.

In general, a query can reference *both* the database tables and the DPViews. Such queries are also issued with the privacy parameter ϵ. We prove below in Theorem 4 that their rewriting guarantees ϵ-differential privacy relative to the authorization policy.

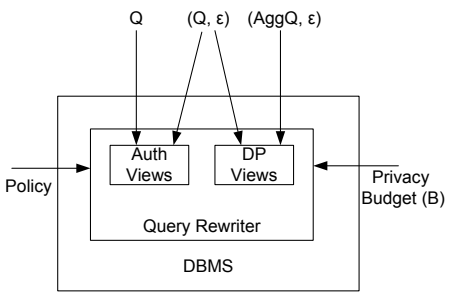


Figure 3: Leveraging Access Control and Differential Privacy

As multiple queries and updates are run, we prove that the system satisfies $\Sigma_i \epsilon_i$-differential privacy relative to the authorization policy where the i^{th} query is run with parameter ϵ_i.

THEOREM 3. *Fix an authorization policy $\mathcal{P}$. Let M_i each provide ϵ_i-differential privacy relative to $\mathcal{P}$. Then the sequence of M_i provides $\Sigma_i \epsilon_i$-differential privacy relative to $\mathcal{P}$.*

PROOF. The proof is almost identical to the analogous previously known result [8] (stated in Theorem 1) — we only have the additional restriction of focusing on instances A and B with $\mathcal{P}(A) = \mathcal{P}(B)$. □

Together with the following straightforward result, Theorem 3 can be used to infer that queries that reference database tables and a DPView with parameter ϵ satisfy ϵ-differential privacy relative to the authorization policy.

THEOREM 4. *Fix authorization policy $\mathcal{P}$. Any deterministic computation that operates on the output of a computation that is ϵ-differentially private relative to $\mathcal{P}$ is also ϵ-differentially private relative to $\mathcal{P}$.*

We also inherit the notion of a per-user privacy budget which is maintained in the same way as in PINQ. The overall architecture is shown in Figure 3.

5.3 Differentially Private Views

We now discuss the class of DPViews we support coupled with how we rewrite them to yield privacy guarantees. The class of queries we encapsulate as DPViews is based on the class of queries for which differentially private algorithms are known [8]. If differentially private algorithms are developed for larger classes of queries in the future, we could correspondingly accommodate a larger class of DPViews.

We note that when the user budget is exhausted, the DPView expression is always rewritten in accordance with the authorization policy while signaling to the application that the privacy budget is exhausted. Thus, the discussion below focuses on the rewriting method for the case when the user budget is not exhausted.

5.3.1 Single Table DPViews

We start our discussion with single-table DPViews. Suppose that a user is only authorized to see a subset of table T. In order to expose privacy-preserving computations on the unauthorized subset of T we can declare a DPView as follows.

```
create noisy view NoisySelect(agg1,...,aggn) as
(select scalarAgg(A1),...,scalarAgg(An)
 from T)
```

We consider standard SQL aggregations namely `sum`, `count` and `average`.

A reference to `NoisySelect` is rewritten as follows if the user privacy budget is not exhausted. We decompose the table T into two parts — the authorized subset and the unauthorized subset. An accurate aggregate is computed over the authorized subset and a differentially private aggregate over the unauthorized subset. The two aggregates are combined. The rewriting for the case where there is only one aggregation and the `scalarAgg` is `count` is shown below in relational algebra-like notation.

$$count(\sigma_{auth(T)}(T)) + noisyCount_{\epsilon}(\sigma_{\neg auth(T)}(T))$$

By Theorems 2 and 4, we can see that the above rewriting guarantees ϵ-differential privacy relative to the authorization policy.

5.3.2 Incorporating Joins

The main challenge in combining joins with differentially private aggregations is that joins have a large sensitivity (recall from Section 4.2.1 that the sensitivity is the maximum influence any single record can have on the output of the aggregation.) Even for the special case of key-foreign key joins the sensitivity can be large since deleting a record from the "key side" can have an unbounded effect on the output join cardinality. Therefore, PINQ essentially only supports key-key joins [8].

In our system, since our goal is to guarantee differential privacy relative to the authorization policy, we *can* support foreign key joins as follows. We introduce the notion of a *stable* transformation relative to an authorization policy.

Definition 4. A transformation T is *c-stable* relative to authorization policy $\mathcal{P}$ if for any two data sets A and B such that $\mathcal{P}(A) = \mathcal{P}(B)$,

$$|T(A) \oplus T(B)| \leq c \times |A \oplus B|$$

Example 2. If we have two tables R and S with R having a foreign key referencing S, then the join $R \bowtie \sigma_{auth(S)}S$ is 1-stable.

If we perform stable transformations relative to an authorization policy before a differentially private aggregation, then the overall computation is differentially private relative to the policy.

THEOREM 5. *Fix an authorization policy $\mathcal{P}$. Let M provide ϵ-differential privacy and let T be an arbitrary c-stable transformation relative to $\mathcal{P}$. The composite computation $M \circ T$ provides $\epsilon \times c$-differential privacy relative to $\mathcal{P}$.*

PROOF. Fix data instances A and B with $\mathcal{P}(A) = \mathcal{P}(B)$. We have:

$$\begin{aligned} & \mathbf{Pr}[M(T(A)) \in S] \\ \leq\ & \mathbf{Pr}[M(T(B)) \in S] \times \exp(\epsilon \times |T(A) \oplus T(B)|) \\ \leq\ & \mathbf{Pr}[M(T(B)) \in S] \times \exp(\epsilon \times c \times |A \oplus B|) \end{aligned}$$

□

We use Theorem 5 to extend the class of DPViews to support foreign key joins as follows.

```
create noisy view NoisyJoin(agg1,...,aggn) as
(select scalarAgg(A1), ..., scalarAgg(An)
from R, S1, ..., Sk
where pJoin and pSelect)
```

In the above expression, the predicate `pJoin` captures the join predicate and `pSelect`, additional selection predicates. We require that each S_i is joined via a foreign key lookup from one of $R, S_1, \ldots, S_{i-1}$ (the intuition behind the requirement is illustrated in Example 2).

We illustrate how the reference to `NoisyJoin` is rewritten when the user budget is not exhausted. We always enforce the authorization on each of the S_i. We decompose the result of $\sigma_{\text{pSelect}}(R \bowtie \sigma_{auth(S_1)}(S_1) \ldots \bowtie \sigma_{auth(S_k)}(S_k))$ into two parts. The *authorized join result* is the subset $\sigma_{\text{pSelect}}(\sigma_{auth(R)}(R) \bowtie \sigma_{auth(S_1)}(S_1) \ldots \bowtie \sigma_{auth(S_k)}(S_k))$ and the rest, i.e. $\sigma_{\text{pSelect}}(\sigma_{\neg auth(R)}(R) \bowtie \sigma_{auth(S_1)}(S_1) \ldots \bowtie \sigma_{auth(S_k)}(S_k))$ is the *unauthorized join result*. We decompose the join into the authorized subset and the unauthorized subset. An accurate aggregate is computed over the authorized subset and a differentially private aggregate over the unauthorized subset. The two aggregates are combined. The rewriting for the case where there is only scalar aggregate namely `count` and no predicates `pSelect` is shown below.

$$\begin{aligned} & count(\sigma_{auth(R)}(R) \bowtie \sigma_{auth(S_1)}(S_1) \ldots \bowtie \sigma_{auth(S_k)}(S_k)) + \\ & noisyCount_{\epsilon}(\sigma_{\neg auth(R)}(R) \bowtie \sigma_{auth(S_1)}(S_1) \ldots \bowtie \sigma_{auth(S_k)}(S_k)) \end{aligned}$$

Again, it is not hard to see that the above rewriting guarantees ϵ-differential privacy relative to the authorization policy.

5.3.3 Incorporating Group By

Suppose that we wish to also incorporate grouping into the class of queries that can define a DPView. Specifically we consider the following expression that extends the DPView `NoisyJoin` above with grouping columns g:

```
create noisy view NoisyGb(g,agg1,...,aggn) as
(select g, scalarAgg(A1), ..., scalarAgg(An)
from R, S1, ..., Sk
where pJoin and pSelect
group by g)
```

Intuitively, we can think of supporting group by by taking each distinct value (group) in the grouping columns and running `NoisyJoin` with additional predicates to select the given group. The first thing to note about this strategy is that a user may not be authorized to see all the groups. So we modify the strategy to only consider *authorized* groups which are the distinct values in the grouping columns in the authorized join result. Second, since the strategy invokes `NoisyJoin` in succession, the privacy guarantee we get is based on Theorem 3. However, since the groups are disjoint we can do better as we show below.

THEOREM 6. *Fix an authorization policy $\mathcal{P}$. Let M_i each provide ϵ-differential privacy relative to $\mathcal{P}$. Let p_i be arbitrary disjoint predicates over the input domain. The sequence of $M_i(\sigma_{p_i}(D))$ provides ϵ-differential privacy relative to $\mathcal{P}$.*

PROOF. The proof is almost identical to the analogous previously known result [8] — we only have the additional

restriction of focusing on instances A and B with $\mathcal{P}(A) = \mathcal{P}(B)$. □

We now describe how a reference to `NoisyGb` is rewritten (when the budget is not exhausted.) The rewriting logically invokes `NoisyJoin` for each authorized group with additional predicates to select the group. However, Theorems 5 and 6 are used to decrement the user's budget only once.

Note that our system is designed such that an unbounded number of queries can be run. So long as the budget permits, the rewriting invokes the unauthorized subset. But after the budget is exhausted, we fall back to basic access control mechanisms. Further, we also note that our semantics can be achieved without any changes to the query execution engine merely by rewriting the reference to the DPView suitably.

5.4 Integrating Parameters

We now sketch one possible manner in which the privacy budget and noise parameters can be integrated into our system. The privacy budget for DBMS users is set as part of the policy specification and managed as a part of the user's metadata. The noise parameter is passed with each query as a connection property. For application users, both the privacy budget and the noise parameter reside in the user application context [11].

5.5 Illustrative Example

We consider a simplified sales database extending the Employee-Department database we have used earlier in the paper. The sales database has the following tables — `Sales(ProductID, EmployeeID, CustomerID, SalesAmount)`, `Employee(EmployeeID, ManagerID)`, `Product(ProductID, Category)`, `Customer(CustomerID, RegionID)` and `Region(RegionID, NationID)`.

The authorization policy lets managers access all records in the `Customer`, `Product` and `Region` tables (note that the `Customer` table in our example does not store sensitive customer information) and the records of employees that are direct reports and their corresponding sales records. Under the above authorization policy, a manager can find the total sales undertaken by each direct report per product through the following query:

```
select E.EmployeeID, S.ProductID, sum(SalesAmount)
from Employee E, Sales S
where E.EmployeeID = S.EmployeeID
group by E.EmployeeID, S.ProductID
```

We note that the above query is rewritten first before execution to only reference the rows the manager is authorized to see. Using the same query above, depending on which manager logs in, we get different rewritten queries (which is the point of supporting authorizations within the database server). In this sub-section, we describe queries as issued by the application noting that they would be rewritten by the system before execution.

Suppose that in order to give a better sense of an employee's sales we wish to compare them with the total per-product sales. The above authorization policy does not grant access to the total per-product sales. We can expose the total per-product sales using the following noisy view.

```
create noisy view NoisyPerProductSales as
 select S.ProductID, sum(SalesAmount) as TotalSales
 from Sales S
 group by S.ProductID
```

Each employee's per-product sales can be compared with the total per-product sales by issuing the following query:

```
select E.EmployeeID, S.ProductID, sum(SalesAmount),
       min(NV.TotalSales)
from Employee E, Sales S, NoisyPerProductSales NV
where E.EmployeeID = S.EmployeeID and
      S.ProductID = NV.ProductID
group by E.EmployeeID, S.ProductID
```

Again, just as with the authorization predicates, `NoisyPerProductSales` is rewritten differently depending on which manager logs in. Thus the output of the query also changes depending on the current user.

Now we consider a different analysis task where a manager wishes to analyze the sales in her department grouped by product category and by nation. Issuing the following query accomplishes this task.

```
select R.NationID, P.Category, sum(SalesAmount)
from Product P, Sales S, Customer C, Region R
where P.ProductID = S.ProductID
  and S.CustomerID = C.CustomerID
  and C.RegionID = R.RegionID
group by R.NationID, P.Category
```

As in the previous case above, in order to compare the sales from a given department with the overall sales while at the same time preserving privacy, we can define the following noisy view.

```
create noisy view NoisyPerProductPerRegionSales as
 select P.Category, R.NationID,
        sum(SalesAmount) as TotalSales
 from Sales S, Product P, Customer C, Region R
 where S.CustomerID = C.CustomerID
   and C.RegionID = R.RegionID
   and S.ProductID = P.ProductID
 group by P.Category, R.NationID
```

The sales within the department and across all departments can be compared by issuing the following query.

```
with DeptSales(NationID,Category,TotalSales) as
(
   select R.NationID, P.Category, sum(SalesAmount)
   from Product P, Sales S, Customer C, Region R
   where P.ProductID = S.ProductID
     and S.CustomerID = C.CustomerID
     and C.RegionID = R.RegionID
   group by R.NationID, P.Category
)
select N.Category, N.NationID,
       N.TotalSales, D.TotalSales
from DeptSales D,
     NoisyPerProductPerRegionSales N
where D.Category = N.Category
  and D.NationID = N.NationID
```

5.6 Summary

The hybrid architecture we sketched above satisfies the noisy view properties outlined in Section 3. We can reduce to the functionality of PINQ using DPViews. On the other

hand, if have only authorization predicates, we reduce to standard access control mechanisms. Further, the examples in Section 5.5 illustrate that we can combine access to authorized and unauthorized portions of data in sophisticated ways, joining them and aggregating them. This is only made possible by integrating the mechanisms of access control and differential privacy (specifically PINQ). In this way, we obtain value beyond the sum of the individual parts. We have also shown that our architecture, while seemingly ad-hoc is in fact a principled approach to integrate differential privacy primitives while preserving its privacy guarantees.

However, we also inherit the baggage associated with existing differential privacy implementations. Therefore, the utility of our framework for complex queries over real datasets remains to be studied. We view our implementation only as a first step in opening up a debate in our community on how database systems can provide meaningful support to address privacy concerns.

6. RELATED WORK

There has been considerable amount of previous work in data privacy [3] and access control [2, 6, 11, 13]. However, to the best of our knowledge, ours is the first paper that studies how access control primitives and privacy preserving mechanisms can be integrated within a database system in a principled manner. The previous work that is most closely related is the Airavat system [14] that combines access control primitives with differential privacy. However, Airavat focuses on the cloud setting where the execution engine is MapReduce. Further, the model of access control considered is mandatory access control rather than discretionary access control supported by the SQL standard and addressed in this paper. Finally, in contrast with Airavat, we formally analyze the privacy implications of combining access control with differential privacy.

Other related work includes techniques for access control over probabilistic data [12] where since the data is probabilistic, the system may not be able to decide whether or not a user has access to a tuple. This is addressed by returning a perturbed tuple — the noise added is such that when it is certain that the user has access, the noise added is 0 and when it is certain that the user does not have access, the value returned is random.

Recent work [7] has addressed setting privacy policies for releasing information about search logs. The privacy policy is implemented using PINQ for differential privacy by setting different privacy budgets for different users. This work is complementary to what we study in this paper.

7. CONCLUSIONS

Data privacy issues are increasingly becoming important for database applications. However the current "black and white" world of access control primitives supported by database systems is clearly inadequate for supporting data privacy. In this paper, we sketch an architecture for a hybrid system that enhances an authorization policy with the abstraction of noisy views that encapsulate previously proposed privacy mechanisms. Accessing data through a set of views is natural for users of database systems and thus the noisy views abstraction represents a natural progression of the concept of authorization views.

We also discuss how we can implement noisy views based on differentially private algorithms. A key advantage of the proposed hybrid system is its flexibility. It can support queries that refer to both the base tables and the differentially private views thus resulting in a system that is more powerful than using access control techniques or differential privacy techniques in isolation. While combining authorizations and differentially private views in this manner seems ad-hoc, we show that it is a principled way to integrate differential privacy primitives with privacy guarantees.

However, our system also inherits some of the limitations of state of the art differential privacy. Therefore, the utility of our framework for complex queries over real datasets remains to be studied. On the whole, we think of our paper as a first step in initiating discussions on how database systems can provide meaningful support to address privacy concerns.

8. ACKNOWLEDGMENTS

We are very grateful to Arvind Arasu for his many thoughtful and insightful comments that have greatly influenced this paper. We also thank the anonymous referees for their valuable feedback.

9. REFERENCES

[1] R. Agrawal, R. Srikant, and D. Thomas. Privacy preserving olap. In *SIGMOD*, 2005.

[2] S. Chaudhuri, T. Dutta, and S. Sudarshan. Fine grained authorization through predicated grants. In *ICDE*, 2007.

[3] B.-C. Chen, D. Kifer, K. LeFevre, and A. Machanavajjhala. Privacy-preserving data publishing. *Foundations and Trends in Databases*, 2(1-2), 2009.

[4] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *PODS*, 2003.

[5] C. Dwork. Differential privacy. In *ICALP (2)*, 2006.

[6] G. Kabra, R. Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *SIGMOD*, 2006.

[7] P. B. Kodeswaran and E. Viegas. Applying differential privacy to search queries in a policy based interactive framework. In *CIKM-PAVLAD*, 2009.

[8] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[9] A. Narayanan and V. Shmatikov. Myths and fallacies of "pii". *Communications of the ACM*, 2010.

[10] A. Netz, S. Chaudhuri, J. Bernhardt, and U. M. Fayyad. Integration of data mining with database technology. In *VLDB*, 2000.

[11] Oracle Corporation. Oracle virtual private database. http://www.oracle.com/technology/deploy/security/database-security/virtual-private-database/index.html.

[12] V. Rastogi, D. Suciu, and E. Welbourne. Access control over uncertain data. *PVLDB*, 1(1), 2008.

[13] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[14] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *NSDI*, 2010.

Transactional Intent

Shel Finkelstein, Thomas Heinzel, Rainer Brendle, Ike Nassi and Heinz Roggenkemper
SAP Research
3410 Hillview Avenue
Palo Alto, CA 94304
{firstname.lastname}@sap.com

ABSTRACT

Data state in a data management system such as a database is the result of the transactions performed on that data management system. Approaches such as single-message transactions and field calls [Gray1993] come closer than before/after values to expressing the intent of a transaction, the semantic transformation that should be performed on the data state even if that state is different than what was previously read. But intent is an even higher-level semantic description. This paper illustrates the use of intent-based transactions and processes in several applications, and describes the benefits from exploiting transactional intent. We provide an application framework for intent, and discuss some advanced aspects of intent, including its relationship to apology-oriented computing [Helland2007].

Categories and Subject Descriptors

H.2.4 **[Systems]**: Concurrency, Distributed databases, Query processing, Transaction processing. H.2.8 **[Database applications]**. J.1 **[Administrative data processing]**: Business, Financial, Manufacturing. D.2.11 **[Software Architectures]**: Data abstraction, Patterns. D.1.3 [**Concurrent Programming]**: Distributed programming.

General Terms

Design, Management, Performance.

Keywords

Intent, data management, database, transaction processing, metadata, business applications, business processes, optimization, supply chain management, supply network collaboration, operational business intelligence, apology-oriented computing, events, callbacks, compensation.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

1. INTRODUCTION

A data management system such as a database (DB) contains data, which at any time has a state. As transactions are performed, that state changes. Database state is an interesting materialized view on the database log. Some systems store multiple versions of some data, in which case all the versions are part of the state. Transactions transform the DB state, by doing insert, update and delete operations (or utilities such as loads) on rows (or sets of rows) in tables.

IMS FastPath [Gray1993] provided single message transactions which transformed IMS state by reading and updating data, only holding locks while the transaction was executed. Field call (e.g., increment/decrement) approaches to state transition have this same transformational character, describing an operation (possibly a program with a series of steps) to be performed, rather than a new state. The advantage of operation-oriented transactions (for transactions consisting of a single operation) is that concurrency control to isolate one transaction from another is only required while the operation is executed, avoiding long-running pessimistic locking as well as optimistic concurrency control rollbacks. (Short-term conflicts can be addressed by retrying operations.) Having transactions consisting of single stored procedures is a generalization of this idea.

Operation-oriented approaches describe state transformations, rather than producing a new state assuming (enforced or hoped for) assumptions about old state matching expectations. The *intent* of a transaction can be a series of operations or a description mappable to a series of operations. If the mapping from a transaction's intent to its state transformation is deterministic, then a log of the intents for a sequence of transactions deterministically defines the transformations performed by the sequence of transactions, based on the composition of (transformations defined by) the intents of those transactions.

As a very simple (and frequently cited) example, consider an inventory transaction whose intent is to change the inventory of a bin based on a formula (such as increment/decrement). The semantics of each operation is well-defined, as is the semantics of a sequence of such operations. Normal execution of a sequence of transactions in a given state produces the intended semantics. However, unless the intent of the transactions is recorded, the general semantics of the operations is lost; they were applied in the current state correctly, but could not be applied in changed circumstances. A transformation from old inventory of 10 to new inventory 0 might have subtracted 10, but it also might have set inventory to 0, or doubled it and subtracted 20.

However, intent may be more semantically sophisticated than just an operation (or a series of operations). The business intent might be to fulfill a sales order from a customer (which happens to be for 10 units of a product), possibly taking into account other customer orders and maximizing some overall goal (such as total profit), a higher level of semantics than the subtraction operation; this intent should be recorded, in addition to operations and/or data transitions.

Why record intent if transactions are complete? As we'll see in the application examples discussed in Section 2, things can go wrong and business circumstances change, so compensating transactions/apologies [Helland2007] may be needed, sometimes including re-execution of transactions in a new state. Intent can be stored in the database just like any other data, allowing it to be searched and accessed (subject to authorization).

Section 3 outlines an application framework for handling intent. Section 4 presents an operational business intelligence example, and section 5 discusses some aspects of business process intent. Section 6 briefly discusses some additional implications and considerations for intent motivated by the examples, and section 7 mentions some related work.

2. APPLICATIONS USING INTENT

This section describes some examples of the use of intent in applications.

2.1 Order Entry

When you deal with an on-line merchant and place an order for the items in your shopping cart (such as books), you often see a screen acknowledging that your order has been received, giving you a number identifying the order. You also often receive an email acknowledging that the order has been received. Before your order is taken, there may be preliminary checking to see if your items are in stock, perhaps by having an (not necessarily up-to-date) inventory estimate in individual order entry nodes (which have catalogs and local transaction histories, which are transmitted to fulfillment services for processing).

Although some order entry systems "guarantee" that the item you ordered is available (e.g., purchases of specific seats), many systems merely acknowledge that they recognize your intent, which is to purchase your shopping cart items. There may be reasons why your purchase cannot be honored, such as too many simultaneous orders of an item (if inventory is not strictly managed).

The separation of order entry from order fulfillment probably seems obvious to people, but teaching clients (humans or applications) to accept that separation is a significant step, separating synchronous capture of the purchaser's intent from an asynchronous response from the merchant system describing actions taken to honor that intent. The response may include the dates items will be shipped and indications that some items may be delayed. If a purchaser specifies that certain items should be shipped together, then delays in one item will delay associated items as well. Only by capturing purchaser's explicit/implicit intent (from order, profile, etc.) can the transaction (or set of transactions) be executed correctly. Even if a subsequent apology is needed (e.g., because a shipment was delayed or a warehouse burned), the same intent-based paradigm is applicable.

2.2 Calendar

This is a description of a hypothetical calendar system using intent; perhaps some system with these capabilities exists.

Suppose that I, as manager, want to schedule a Project Review meeting. It must include the project lead and at least 3 out of 4 staff members, and it must be after an architectural review meeting. The value of the meeting is higher the earlier in the week it is scheduled, but it must be scheduled before next Friday.

This is an intent description that captures constraints and informal objective functions ("earlier in the week is better") for my meeting. If the meeting were scheduled for Tuesday at 2pm without capturing the intent, then it wouldn't be possible to automatically reschedule the meeting if the Architectural Review were delayed. Rescheduling might involve moving a lower priority meeting for me or one of the other staff members.

2.3 Supply Chain Production Scheduling

SAP Advanced Planning and Optimizer (APO) [Balla2007] does planning tasks, such as scheduling machine runs (production orders) to produce finished products. Machines have given capacities; jobs require materials (which may have to come from other jobs). Constraints (e.g., latest delivery dates) and objective functions (weighted combinations of time and costs) are specified, and APO optimizes scheduling to maximize (heuristically) the objective function while meeting the constraints.

Requests to APO model intent either to produce product for inventory stock, or to deliver products for customers. APO acknowledges these requests. Production jobs to fulfill the requests may be scheduled incrementally based on requests taking into account existing resource schedules. Periodically, global optimization may be performed across all jobs. Because intent is captured, not just proposed job fulfillment schedules, such incremental and global scheduling is possible across all jobs, adjusting schedules when necessary for new high priority jobs, which may delay previously entered lower priority jobs.

There is also intent (which we call meta-intent in section 6.3) in the way administrators define master data for the scheduling algorithm, so that certain customers, products, locations and organizations have high priorities.

2.4 Supply Chain Business Interactions

SAP's Availability-to-Purchase (ATP) [Balla2007] is part of Supply Chain Management. To initiate a purchase, a purchaser issues a request for merchandise to a supplier, specifying quantities, delivery dates, quality, etc. The supplier responds with a term sheet that may specify multiple alternatives for quantities, dates and prices of items that are available to purchase, as well as term sheet acceptance deadlines. The purchaser may submit a purchase order based on the term sheet, and the supplier can

create an internal sales order and schedule deliveries after the purchase order is accepted.

Each of these steps involves an expression of intent between purchaser and supplier, which enables parties to deal with exception cases and issue/handle apology events. For example, while waiting for the actual purchase order after sending a term sheet, the supplier might choose to reserve quantities only for a limited time (or not at all), and the supplier might have to apologize to the purchaser if quantities are not available after the purchase order is submitted. Later in the process, a supply delivery may be delayed due to a manufacturing problem, just as merchandise delivery may be delayed in the order entry example. Term sheets (describing quotations for items available to purchase) may become invalid for business reasons, e.g., because the purchaser is no longer eligible for discounts or because they time out. Capturing intent supports both better optimization across the set of intents for all sales order, not just based on current schedules, as well as compensation actions (which are forward actions, not rollbacks) if constraints for a particular sales order no longer can be met due to higher priority sales orders.

2.5 Supply Network Collaboration

Applications such as SAP's Supply Network Collaboration (SNC) [Hamady2009] handle collaborative negotiations between companies. For direct material replenishment, price is negotiated up-front but terms for delivery times and quantities can frequently change from both demand-side and supply-side. SNC records current values and past histories for interactions between purchasers and suppliers. A supplier may send an offer to a customer, expressing terms to sell; a purchaser may send an offer to a supplier, expressing terms to buy. These are independent, offers. Either party may propose an agreement based on the terms expressed in the other's offer; for example, the purchaser may send an offer to buy on the supplier's terms. The supplier decides whether to confirm the sale on the terms that it offered; business conditions may have changed. If the supplier confirms, then there is an agreement… unless one of the parties subsequently cancels, requiring handling of that cancellation by the other party.

The communicating purchaser/supplier state machines for SNC record and exploit intent for both business parties (supplier and purchaser). For example, if a supplier does not confirm an agreement, or reneges on terms, the purchaser can determine what the intent of the purchase is, and decide whether to pursue new terms with that supplier or another supplier. Purchaser-side event handlers (which are rule-based) determine which deviations in supplier confirmations should automatically be accepted, and which a human being needs to review.

If the purchaser only knew about the planned delivery (data derived from negotiations) or about the logical operation performed (acceptance of supplier's terms), there would not be enough information to handle the cancellation event (apology) properly. Knowing the purchaser's intent to obtain delivery of merchandise by a given date enables the purchaser's system to deal with the cancellation. Mechanisms and data for doing this are discussed in the next section.

3. APPLICATION FRAMEWORK FOR INTENT

Applications and application frameworks exploit the capabilities of the data management layer of a system, but are not themselves part of the data management layer. Delivering transactional intent for the applications described in section 2 requires an application framework supporting intent. This framework can be built on top of existing data management functionality, although optimizing data management for intent may be valuable, particularly for some of the more advanced capabilities described in section 4.

This section gives a very informal description of what intent is and what the requirements are for an application framework that handles intent. There are many alternative ways that intent can be expressed formally, and many frameworks that meet these requirements, and a single business application suite can support multiple alternative implementations.

3.1 Intent expressions

Definition: **Intent** is an expression of the **goals** and **constraints** that should be met by a business transaction or business process, optionally with **optimization parameters** (e.g., objective function parameters or priorities). There also can be **satisfaction events/callbacks** associated with satisfying the intent and failing to satisfy the event, both when the intent is initially satisfied/not satisfied, and when there are changes in how/whether the intent is satisfied.

Intent for a given problem domain implementation may be expressed using a domain specific language. The expression of intent may be imperative (e.g., explicit code to subtract 10 from Inventory as long as result is zero or more), declarative ("schedule a meeting having the following participants, occurring after an architecture review meeting but before Friday") or some combination of declarative and imperative. The only requirement is that intent be "understood" by the application framework intent optimization engine.

3.2 Intent optimization engines

An application framework for executing intent includes an **intent optimization engine** that determines which intents will be satisfied and how such intents will be satisfied (intent execution plans). Intent may be regarded as metadata; it doesn't describe what has happened (data) or what is going to happen (plan or projections); instead, it describes what the **intent submitter** would like to happen. Any intent execution plan that achieves the intent's goal while meeting its constraints satisfies that intent, and should be acceptable to the intent submitter. But the intent optimization engine heuristically optimizes (based on objective functions or priorities) across all intents.

For example, an intent optimization engine for Supply Chain Production Planning would schedule materials and machine runs to produce products (production orders). Optimization parameters such as priorities and objective function parameters are used by the optimization engine to make scheduling decisions. A scheduling engine that chooses intents based on highest priorities

would expect priorities as its optimization parameters. Other intent optimization engines might maximize an objective function total across all intents whose constraints are met, perhaps subtracting penalties for intents that cannot be satisfied. For example, Supply Chain Product Planning might maximize total revenue or profit. More complex optimization strategies are also possible, such as maximizing profit while ensuring that all high priority jobs are finished on time. On the other hand, a very simple optimization engine could use a rule-based approach to satisfy constraints without using priorities or objective functions.

Some optimization parameters might be administratively determined, such as the business importance of the user, organization or the process instance submitting the intent. Other optimization parameters, such as deadlines, and the penalties for not meeting an intent's deadline, might be explicit optimization parameters, or could be expressed as metadata.

When a new intent is submitted to an intent optimizer engine, one scheduling approach is to find the best way to satisfy the new intent without disturbing existing plans. Another approach is to satisfy new high-value intents (with high priority or large objective function contributions) with minimal disruption by rescheduling the plans of one or more intents with lower priority. Of course, such incremental heuristic approaches may not find the maximum utility schedule as determined by the objectives function, so periodic global optimization may be appropriate. In Supply Chain Production Scheduling, global optimization can be expensive when there are many (tens of thousands) resources, products and jobs/intents. Dealing with "apologies" due to rescheduling is another expense, even when products are produced sooner than expected. Tradeoffs between incremental and global optimization depend on the individual organizations and applications involved. Another common scheduling approach (which has both advantages and disadvantages) is to partition the problem into smaller sub-problems which are addressed independently, e.g., by first assigning a job to a particular factory and then doing scheduling within that factory.

3.3 Satisfaction events/callbacks

When an intent optimization engine initially determines how an intent will be satisfied (or that it won't be satisfied), it generates a satisfaction event that notifies the intent submitter (and other interested parties or authorized event subscribers) about its decision. For example, when a calendar appointment is scheduled, people invited to the meeting should receive meeting requests. A simplified approach, which we'll emphasize, requires an intent submitter to specify satisfaction callbacks rather than satisfaction events. Note that there can be separate satisfaction and non-satisfaction events (or callbacks) associated with a given intent. If an appointment's meeting time has to be adjusted subsequently due to a scheduling conflict such as a more important meeting, then a new satisfaction event/callback would be generated. If the appointment has to be cancelled, or delayed past its deadline, than a non-satisfaction event or callback is generated. Such events/callbacks are usually not handled within the transaction generating them, but the framework should guarantee that they will be executed once and only once (using well-known retry and idempotence techniques [Gray1993]).

The event associated with rescheduling or cancelling a delivery or a calendar appointment has been called an apology [Helland2007]. Such events may be frequent in flexible planning environments requiring frequent intent re-optimization. When a purchasing process in Supply Network Collaboration receives a cancellation apology, it must determine how to cope with that event.[1] The purchasing process would compensate by marking the previous agreement as cancelled (and perhaps take other actions, such as tracking supplier's reliability), and then the purchasing process would find another way to meet its intent to received the merchandise by the given date, perhaps by ordering from a different supplier.

3.4 Change metadata

To perform compensation, the purchaser process must know **change metadata** associated with intents (or more specifically, with a plan to satisfy intents), and then perform application-specific methods for compensation, which are always forward-going set of actions, not rollbacks. Change metadata includes three elements:

1. Intents and the **intent execution plans** for meeting those intents, which may involve business objects and sub-intents.
2. A **dependency graph** between business objects, a directed graph labeled with callbacks[2] and conditions in which those callbacks should be invoked. When there's an edge between objects A and B, the callbacks are on object B, while the conditions may involve both A and B.
3. **Version history** for objects, indicating not only value changes but also the intents that led to those changes.

Change metadata describes directed cross-relationships among object versions and intents. As we'll see, these relationships may have been created in different transactions, and even by different business processes running different applications.

3.4.1 Change metadata examples

Example: In the calendar application, meeting M2 may have to occur after another specific meeting, M1, so meeting M2 depends on the timing of meeting M1, and there's an edge between M2 and M2. If M1 is moved to a later time, then the callback on M2 associated with the (M1, M2) dependency edge is invoked.[3] The intent which created M2 (identified in M2's version history)

[1] A purchasing process may also receive notice that a supplier has reduced its confirmation level, which is a weak form of cancellation.

[2] The dependency graph can also be viewed as describing subscriptions to object change events, but we'll mainly use the callback approach in this section.

[3] A more sophisticated implementation might invoke that callback only if M1 now occurs after M2; a less sophisticated implementation might invoke that callback if M1 changes in any way, not just its timing.

might have specified a deadline. If the change in M1 means that M2 can no longer be scheduled before its deadline, then a non-satisfaction callback for M2 will be invoked. The person who scheduled the meeting (or a human or software agent for that person) will be notified and make take further actions, such as moving a higher priority meeting so that M2 can occur on time.

Example: A sales order may have been created in an Availability-to-Purchase application. In Supply Chain Production Scheduling, the dependency graph may include an edge going from that sales order to a machine production order that was created to help fulfill that sales order. This edge indicates that the production order depends on the sales order. If the sales order is cancelled, then the machine production order should also be cancelled, or perhaps redirected to fulfill another sales order, which results in a different dependency graph edge.

But the sales order also depends on the machine production order, so in this case there are edges in both directions.[4] If the machine order cannot be completed because a production line machine failed, then the intent execution plan for the sales order needs to be revisited. The execution plan for the sales order may involve multiple production orders on different machines, each with a sub-intent created to fulfill the overall intent of fulfilling the sales order. The sub-intent behind the failed machine production order could be fulfilled using other production line machines, as long as scheduling dependencies among jobs (which create outputs used by other jobs) are met.

3.4.2 Storing /maintaining change metadata

As the production scheduling example above shows, objects created by one business application may depend on objects created not just by other transactions and business processes but even by other business applications. Hence the change metadata tracked, the means of tracking it, and the methods for handling callbacks require that the set of applications cooperate in a common framework. Let's discuss storing and maintenance of the three elements of change metadata.

3.4.2.1 Storing/maintaining intent execution plans

Storing intent execution plans is relatively straightforward. When an intent execution engine creates a plan satisfying the intent, it stores the intent and the execution plan associated with it. A plan may be hierarchical, involving satisfying sub-intents (such as individual product orders used to satisfy a sales order), each of which has an execution plan. When a plan is updated due to a satisfaction event/callback, the new plan is stored, perhaps keeping the old version (and the reason it was updated) for auditing, historical data mining and predictive analytics.

[4] Although there are edges in both directions, a change in one object won't lead to an infinite loop because changes damp out. Infinite loops should be highly unlikely in correct programs because of application semantics.

3.4.2.2 Storing/maintaining dependency graphs

The dependency graph is more difficult to maintain because it is a cooperative data structure built across many instances of different business processes and applications. In a sense, the dependency graph helps unify different applications. An application may access/update many objects; the application writer (or maintainer) must understand which inter-object dependencies matter, as well as how they should be addressed. This knowledge is specific to each application domain, but fortunately that knowledge is often separable by application.

A developer who is writing (or changing) application X should understand which business objects X depends on, as well as how other objects in X depend on those objects. Objects in X may depend on other objects in X (e.g., a production order dependent on other production orders) or on objects that are created or modified outside application X (e.g., a production order dependent on a sales order). Although a developer writing or modifying X[5] may need to know how the objects that X depends on could change (so that X may react to those changes),[6] the developer need not know the internals of the applications that change the objects X depends on. However, the intent of the changes made by those other applications may be worth knowing (and is available in version history, describe in the next subsection).

Writing or maintaining an application requires creating and maintaining the dependency graph, including the callbacks (and code for handling the callbacks) associated with it. Although there could be tools that help with dependency graphs and application separability helps simplify the problem, the creation and maintenance of dependency graphs requires application domain knowledge and some stylistic conventions. For example, if a production order depends on a particular sales order, then including the sales order as a parameter for functions creating or modifying the production order helps the developer describe the dependency.

3.4.2.3 Storing/maintaining version history

Version history, like intent execution plans, is relatively straightforward. The version history for business objects is similar to versioned data in databases, although it has additional information supplied by the application framework. That additional information could identify the intent of the transaction that created each data version, and provenance-like descriptions of subparts of transactions (like production orders created to fulfill a sales order), reflecting their sub-intents. If intent is stored for each transaction, then storing the transaction id for each version

[5] Writers of specific components of application X should only need to know about object dependencies involving objects in their components.

[6] One coarse way a developer can deal with updates of objects you depend on is to treat each of them as a deletion followed by an insert, which simplifies the set of changes the developer must handle. In practice, that would be an awkward and inefficient approach, hiding the semantics of the update and potentially obfuscating its intent.

identifies the intent of the transaction, but doesn't identify higher level business process intent (discussed in section 5) or lower level sub-intents, such as creating production orders.

Additional "provenance" that could be stored in version history is the events/callbacks processed due to dependencies, including the object that changed, the dependent objects and actions taken by callbacks (which typically involve one or more transactions and intents). As with intent execution plans, keeping versions of this data may be valuable for auditing, historical data mining and predictive analytics.

3.5 Some application framework implementation considerations

In this section we informally described an application framework for intent, including intent expressions, intent optimization engine, satisfaction events/callbacks and change metadata. Change metadata is the most complex part of this, particularly dependency graphs.

We emphasize that the capabilities described in this section are all at the application layer (application framework plus application programming), although the application layer leverages data management capabilities to retrieve and update application data and metadata. Of course, transactional techniques should be used to ensure atomicity of data/metadata updates, as well as the asynchronous events they generate (which are handled outside the transaction). SAP traditionally has handled many aspects of data management (buffering, locking, update queues) in an application layer outside of the database [Finkelstein2008], only touching the database (beyond reads) when transactions commit, and SAP implements some aspects of the application framework we described in that application layer.

Stored procedures can improve performance and encapsulation by executing code within that data management system. For example, a scheduler could select intents to satisfy, and then schedule them using a single transaction running as stored procedure at an appropriate isolation level. This reduces pessimistic lock duration significantly, or reduces rollbacks if optimistic concurrency control is used in the DB.

Putting too much code within the DB might make it a bottleneck, and would reduce the flexibility of the application layer. Balancing database and application tier capabilities is an art with some challenging tradeoffs worth further consideration. We're interested in application (and application framework) specification paradigms that support multiple execution bindings to the application and database layer, with "best" binding selected by an optimizer.

4. OPERATIONAL BUSINESS INTELLIGENCE

In this section, we describe another example, an approach to operational business intelligence (OBI) using intent that we think may (appropriately) do a better job of delivering the actual intent of decision-makers better than some other approaches. We'll explain the problem, then talk about OBI with a single database, and finally look at OBI with multiple databases. That leads us to business process intent, which is discussed more generally in the section 5.

4.1 The operational business intelligence problem

When decision makers or other knowledge workers see reports generated from databases, they may detect problems in their businesses which they want to address. Something might be wrong that needs to be fixed, such as a (repeatedly) broken production line, or there might be a better way of handling an issue, such as a customer escalation. Business intelligence involves getting data from one or more databases so that decision makers can act on it in a timely way. At one time such reports were generated weekly or nightly, but decision makers increasingly want such information immediately, using current (or nearly current) data. Instead of requiring decision makers to ask the right questions, active databases may automatically alert people when unusual events (or unusual complex events, composed from other events, e.g., a sequence of events) occur. This allows people (or automated agents) to react quickly to these unusual events, handling them as soon as possible.

However, even if corrective actions are taken very quickly, the state of the database that the decision maker observed may have significant differences from the state of the database when action is taken. The production line might be under repair by a restart or other action that's underway. Another decision maker (human or automated) might have addressed the production line failure by diverting some other production line to deliver high priority products assigned to the failed production. An appropriate corrective action in the original state might be completely inappropriate in the current state, whether it's minutes, seconds or even fractions of seconds later.

Moreover, the state of the database and the state of the real world aren't necessarily in agreement. The observation that the production line was broken might be erroneous. Other production lines may have failed, so an appropriate action with only one production line down might no longer be appropriate. Even if the database state and the real world are in agreement, good and bad things might happen subsequently, such as a new production line starting or power going out in a factory. If the only information that is captured from the decision maker is the decision they requested, or worst yet, the data operations (and associated real world actions) they caused, then it seems impossible to deal with a situation different from what was expected when the decision was made.

The operational business intelligence (OBI) problem is determining how to use business intelligence to make operational business decisions that are appropriate for both the decision time and the current database state when the decision is applied. That is, decisions should continue to be applicable by application code in appropriate ways when the database state changes.

4.2 Intent and operational business intelligence for one database

Not surprisingly, we believe that intent is an excellent approach for OBI. If the intent of a decision is captured, not just a set of actions that effectuate that decision, then the intent optimization engine can try to meet that intent in any current (or future) database state.

For example, if a decision maker believes that a production line manufacturing a product for a sales order has failed, and product delivery is more important than was previously indicated, then the decision maker could increase the priority of that production job or of the sales order that generated that production job. If an objective function rather than priority is used to schedule machine production lines, then the decision maker could bump up the value of servicing this particular customer or this particular sales order.

If the production line has recovered, or if another decision maker has taken actions to ensure that this production order will complete, then the original decision maker's action will have no effect since their intent was already being met. The intent optimization engine might even be taking action to handle the production line failure already, based on the intent originally expressed for the sales order. Changing the priority of this production job might cause it to finish more quickly, or might have no effect.

Some decision makers might not understand detailed priorities and objective functions, and the impacts these could have on production line scheduling. It would be better if the user experience for decision makers were expressed in their own terms, where they could request (or require) that job constraints be met, possibly with coarse granularity priorities. Alternatively a rules-based approach could be used. Decision makers should also be able to view the intents that can be satisfied and the intents that cannot be satisfied with visualizations that match their roles and experiences. For example, if an intent cannot be met, then a summary visualization of higher priority intents might be visually presented showing revenue, responsible groups and priorities for those intents. For optimization algorithms, this can be achieved with explanation components that explain the constraints, and ideally also explain the contribution of various elements to the target function.[7]

4.3 Intent and operational business intelligence for multiple databases

Now let's consider the operational business intelligence problem when business intelligence data may come from multiple database sources, either via a data warehouse or via queries to these databases. For simplicity, we'll assume that there are two databases. At first glance, this may seem like the same problem that we considered in the previous section. If actions can be taken against both databases using distributed transactions with two-phase commit, then the problem is essentially the same as in the previous section. However, there are good reasons, such as performance and failure handling [Helland2007], for avoiding two-phase commit unless both databases are in the same management domain, and perhaps even when they are.

Let's assume that distributed commit is not acceptable, and the decision maker wants to take actions that affect both databases. Since we've assumed that two-phase commit isn't acceptable, we can't perform actions against the two databases atomically. Instead, however, we can perform actions similar to those in long running transactions and sagas [Gray1983, Garcia-Molina1987], where transactions are performed against both databases, and both must succeed for the saga process to complete successfully. A common example is planning an itinerary for a trip to a city, requiring both a roundtrip flight and a hotel, which are booked in different databases. If a traveler books a flight but can't get a hotel for those dates, then the flight will be cancelled, and the user may try to book the trip on different dates (or to a different city).

In an OBI situation, the traveler may see flights and hotels (and their prices) together in a warehouse, and may decide to book a particular flight and hotel. Using intent, the traveler could express date, price and hotel location constraints, as well as an objective function that trades off price with flight duration and hotel quality. The intent optimization engine could choose a flight and hotel for the traveler, and might book the flight, only to discover then the hotel no longer has rooms available for those dates. It might choose another hotel room, or might cancel the flight and select hotel and flight on another date. In some ways, this is similar to the calendar example from section 2, except that multiple databases are involved, so an application level business process is required, not just a single transaction.

If a room at a better hotel becomes available cheaply, then a satisfaction callback would be executed allowing the traveler to switch hotels. If staying at that hotel requires changing flight dates, then the change should only happen if the objective function (including flight change penalties) improves; the change in flight and hotel should be made carefully, so that the traveler doesn't release existing reservations until the new reservations are confirmed. Other changes, such as a hotel that closes down or a travel date change request made by the traveler, could also be addressed by a travel application based on the traveler's intent.

5. BUSINESS PROCESS INTENT

Most of the infrastructure and examples that we've discussed so far in this paper have involved use of intent for transactions, and the title of this paper is "Transactional Intent". However, intent can also be used for business processes, as illustrated by the Supply Chain Business Interactions and the Supply Network Interactions examples in section 2, as well as by the multiple database travel itinerary example in the previous section.

In this section, we first describe a practical basic approach to decomposing business process intent into transactional intents, and then mention some intriguing aspects of intent (internal, externalized and predictive intent) for multi-party processes.

[7] See, for example, APO Supply Network Planning, http://help.sap.com/saphelp_ewm70/helpdata/en/87/383e4229f1f83ae10000000a1550b0/content.htm

5.1 Intent decomposition

Addressing business process intent requires decomposing the business process and its intent into subparts (such as transactions), each of which has its own intent. Sometimes the decomposition into transactions and associated intents may seem clear from the definition of the process. For example, the travel itinerary example may be decomposed into multiple transactions in more than one way (get flight first and hotel second, or get hotel first and flight second). The transactional intents for the "flight, then hotel" decomposition would be "Find best flight meeting constraints" and then "Find best hotel meeting constraints, including added constraints imposed by the flight". But even this simple example raises questions: How were these decompositions identified? Do other plausible decompositions exist?[8] How were the transactions and transactional intents in the decomposition determined? In the example, how were "the added constraints imposed by the flight" identified?

Solving the general intent decomposition problem resembles solving a general bottom-up goal-directed artificial intelligence problem. We're not going to try to address such general automatic programming problems in this paper. Practical systems are more likely to have one or more decomposition patterns identified for each business process, describing the transactions and their intents, as well as other application framework aspects (e.g., satisfaction callbacks) described in section 2.

The transaction and intent decomposition patterns for the Supply Chain Business Interactions and Supply Network Interactions business process examples presented in section 2 are more complex and more flexible, involving not only transactions but also messages between parties, where each message expresses intent. However, there typically are standard decomposition patterns for these business processes; whenever the business process initiates a transaction, it can associate an intent with that transaction, while also identifying itself and its own intent.

5.2 Multi-party business processes

Processes which involve multiple parties introduce some new aspects because the two (or more) parties involved may not fully share their actual intents with each other. A purchaser may request a term sheet from a supplier, but its intent may be to pressure other suppliers to lower their bids. We can distinguish among the following types of intent in a two party interaction between purchaser and supplier:

- The purchaser's intent, known only by the purchaser, an internal intent.
- The supplier's intent, known only by the supplier, also an internal intent.
- The intent which the purchaser expresses in messages to the supplier, an externalized intent.
- The intent which the supplier expresses in messages to the purchaser, also an externalized intent.
- The intent which the purchaser infers from the supplier's past and current behavior, a predictive intent.
- The intent which the supplier infers from the purchaser's past and current behavior, also a predictive intent.

The supplier can act based on its own internal intent, the purchaser's externalized intent, and the predictive intent that the supplier has inferred from the purchaser's behavior; a similar statement can be made for the purchaser.

Behind the internal intent expressed in the purchaser and supplier software, there is also the intent of the human beings (if any) making decisions during the business process. Predictive intent analyzes externalized behavior to try to infer intent, but that intent could depend on the specific human beings making process decisions. One reason to store version histories including intents is to help with such predictions, although different people may have different intents and behaviors.

6. INTENT CONSIDERATIONS AND IMPLICATIONS

Section 2 described some applications that use (or could use) intent. Although many of the ideas in this paper have been in use in systems for many years, and others ideas, such as apologies, have been proposed before, the conceptual framework described in section 3 is novel, describing a high-level architecture for handling intent. Section 4 described use of intent for operational business intelligence, and section 5 discussed some aspects of process intent. In this section we present some additional considerations and implications, which we hope will lead to future work on transactional and business process intent.

6.1 Auditing, data mining and predictive analytics

Applications following the framework described in section 3 store a lot of data and metadata about intent, including data versions, intents and sub-intents, execution plans for meeting intent, satisfaction events/callbacks that induce re-planning, dependency graphs and other intent metadata. This information is valuable for many reasons; it can be used to explain decisions to users and partners, as well as for auditing and regulatory requirements. Moreover, it may be valuable for data mining and predictive analytics concerning decisions. For example, a supplier who frequently reneges on deliveries could be regarded as a risky choice for future critical orders, even if that supplier's terms are excellent. Predictions are useful even for single party processes, e.g., to predict whether a production run is likely to complete before its deadline during a busy time of the month. More generally, the higher level semantics expressed in intent may enable deeper data mining and richer predictions than can be made based only on data history.

[8] One such decomposition involves selecting multiple good flights and hotels independently in parallel, finding the best matching pair, and then reserving that flight and hotel, if possible, trying again if it's not.

6.2 Compensation and change

Utilizing the framework described in section 3, intent supports compensation when a submitted request can no longer be completed as originally planned. Intent can be the basis for a process exception management approach that is usable either within a single company or in B2B contexts, since it captures semantics and metadata which value and operation-based approaches omit. Instead of compensating for a failed plan and then building a new intent execution plan, these two steps could be correlated and combined, so that the old plan is modified rather than deleted and replaced. When it's workable, this approach could be relatively efficient. For example, dependency graph edges that exist in both old and new plans might be retained, rather than compensated for and then recreated.

6.3 Meta-intent

Intent may be expressed using imperative or declarative approaches (e.g., with declarative constraints). Intent optimization engines also may be written using imperative or declarative models, or a combination of both. Using a declarative engine makes it easier to specialize incremental or global intent optimization algorithms to meet specific circumstances. Let's consider a supply chain production scheduling example. One manufacturer may have particular policies and algorithms for Advanced Planning and Optimizer (APO) that are different than those for other manufacturers; these policies and algorithms capture that manufacturer's **meta-intent**, that is, his intent in optimizing the intents of his users.[9] (APO was described in section 2.3.)

Intent scheduling may also be polymorphic for a particular manufacturer, with different scheduling algorithms used for different classes of users or products. As usual, declarative encapsulation makes it easier to introduce, modify or replace such algorithms.

6.4 Eventual consistency

Intent can help deliver eventual consistency [Vogels2008] for replicated data based on ACID 2.0 (associative, commutative, idempotent, distributed) [Finkelstein2009, Helland2009]. For example, for scalability, availability and locality, there might be multiple sites that handle Advanced Planning and Optimizer (APO) requests and schedule jobs for the same factory, with job data (unique job names and job intents) replicated asynchronously among the sites. This approach could also be used for eventual consistency of multiple order entry sites, or for disconnected mobile calendars, which are much more likely uses than APO.

Assume that:

a) all job data eventually arrives at all sites, and

b) global scheduling is deterministic, based only on the set of distinct jobs, not on their order of arrival.

Then the job data at all sites will eventual converge, and the job schedules at the sites will eventually be consistent. (Timeliness is an important issue that we won't address here.) At each site, this approach to eventual consistency uses the application framework (with intent satisfaction events and callbacks) described in section 3. It does not require distributed transactions or complex replication protocols, only conditions a) and b).

6.5 Cooperating businesses and applications

Business process intent is a particularly important tool at the boundaries between worlds described by different applications and systems. For example, intent helps synchronize plans across customers and suppliers in a multi-tier supply chain, where satisfying an intent request from X to Y may depend on satisfying an intent request from Y to Z.

Intent could also be used to align the delivery schedules of multiple distribution centers run on different systems, so that they can cooperate to serve worldwide product demand.

6.6 Performance and usability

Used badly, intent may engender poor user experience and heavy system load. For example, if global optimization of a large APO system is performed every time a new purchase order is created, the performance load and rescheduling instabilities could be unreasonable. Like other optimization schemes, intent optimization needs to be properly calibrated; doing incremental optimization or partitioned optimization (over a related subset of orders) might be a better approach than continual global optimization. Global optimization could still be performed, but only infrequently, and its potentially disruptive results might be adopted only when they substantially superior to existing schedules.[10]

Other user experience issues include entering intent and writing applications using intent. We discussed ways that decision makers might enter intents in their own terms, perhaps using a "visual intent" interface. Programming applications using intent has many non-trivial aspects (such as dependency graphs) where tools, libraries and programming conventions would help. But we believe that systematic use of intent can make application programming easier, requiring less expert knowledge across different application domains.

7. RELATED WORK

The most closely related work that we know of is Helland's apology-oriented computing [Helland2007], which we've already discussed. This paper generalizes ideas in apology-oriented computing and proposes a framework for the generalization.

[9] Configuration and customization of an installation are metadata operations which implicitly or explicitly capture aspects of the installation's intent in processing requests, as opposed to the intent of a particular request.

[10] This destabilization problem may be a reason that calendar systems don't follow the approach suggested in section 2.2.

Field calls and commutative locks/escrow locks were mentioned in the introduction of this paper and are discussed in [Gray 1993], but intent is at a very different semantic level.

Semantic data types are discussed in papers including [Schwarz1984, Weihl1988]; intent uses semantics, but in a different way and at a different level.

There have been many papers on long running transactions; a number of early works are cited in [Gray1993], including Garcia Molina and Salem's work on sagas with compensating transactions [Garcia-Molina1987]. Our paper uses both ideas, long running transaction and compensation, but creates a novel framework using them.

Operational transform [Ellis1989] has been discussed recently because of its use in Google Wave. As with intent, the goal is to determine what transaction to perform in a changed state. Operational transform achieves this either because operations (such as appends to a thread) commute, or by inferring the operation from a state transformation. Intent does not require operation inference; it explicitly represents requested semantics, and it does this for a transaction, not just a single operation.

A number of papers (e.g., [Agrawal2009, Sadikov2010]) have examined the problem of meeting the intents of users performing web search, but although the word "intent" is used, these papers are addressing a very different problem than we are. Our predictive intent has an indirect connection to web search intent because both involve mining and prediction.

8. REFERENCES

[Agrawal2009] Agrawal , R., Gollapudi, S., Halverson A. and Ieong, S. 2009. Diversifying Search Results, Proceedings of the Second ACM International Conference on Web Search and Data Mining.

[Balla2007] Balla, J. and Layer, F. 2007. Production Planning with SAP APO-PP/DS, Galileo Press, Boston, MA and Bonn Germany.

[Ellis1989] Ellis, C.A. and Gibbs, S.J. 1989. Concurrency Control in Groupware Systems. ACM SIGMOD Record 18 (2), p.399–407.

[Finkelstein2008] Finkelstein, S., Brendle, R., Hirsch, R., Jacobs, D., and Marquand, U. 2008. The SAP Transaction Model: Know Your Applications, presented (but not published) at ACM SIGMOD 2008 Product Day, Vancouver Canada.

[Finkelstein2009] Finkelstein, S., Brendle, R., and Jacobs, D., 2009. Principles for Inconsistency, Proc. CIDR, 2009.

[Garcia-Molina1987] Garcia-Molina, H. and Salem, K. 1987. Sagas, Proc. SIGMOD Conference 1987, p. 249-259.

[Gray1993] Gray, J. and Reuter, A. 1993. Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Mateo, CA.

[Hamady2009] Hamady, M. and Leitz, A. 2009. Supplier Collaboration with SAP SNC, Galileo Press, Boston, MA and Bonn Germany.

[Helland2007] Helland, P. 2007. Life Beyond Distributed Transactions, An Apostate's Opinion, Proc. CIDR 2007.

[Helland2009] Helland, P. and Campbell, D. 2009. Building on Quicksand, Proc. CIDR, 2009.

[Sadilov2010] Sadikov, E. , Madhavan, J., Wang , L., Halevy, A. 2010. Clustering Query Refinements by User Intent, Proc. of the 19th International Conference on World Wide Web.

[Schwarz1984] Schwarz, P.M. and Spector, A.Z. 1984. Synchronizing Shared Abstract Types, ACM Transactions on Computer Systems (TOCS), v.2 n.3, p.223-250.

[Vogels2008] Vogels, W. 2008. Eventually Consistent, ACM Queue, 6(6), p. 14.

[Weihl1988] Weihl,W. 1988. Commutativity-Based Concurrency Control for Abstract Data Types, IEEE Transactions on Computers, v.37 n.12, p.1488-1505.

Consistency in a Stream Warehouse

Lukasz Golab and Theodore Johnson
AT&T Labs – Research
180 Park Avenue, Florham Park, NJ, USA 07932
lgolab@research.att.com, johnsont@research.att.com

ABSTRACT

A *stream warehouse* is a Data Stream Management System (DSMS) that stores a very long history, e.g. years or decades; or equivalently a data warehouse that is continuously loaded. A stream warehouse enables queries that seamlessly range from real-time alerting and diagnostics to long-term data mining. However, continuously loading data from many different and uncontrolled sources into a real-time stream warehouse introduces a new consistency problem: users want results in as timely a fashion as possible, but "stable" results often require lengthy synchronization delays. In this paper we develop a theory of temporal consistency for stream warehouses that allows for multiple consistency levels. We show how to restrict query answers to a given consistency level and we show how warehouse maintenance can be optimized using knowledge of the consistency levels required by materialized views.

1. INTRODUCTION

Many real-world enterprises generate streams of information about their operations and require real-time response for their maintenance. Examples include financial markets, communications networks, data center management, and vehicular road networks. Data Stream Management Systems (DSMSs) have been developed to provide real-time analysis and alerting of these and other data streams, typically by processing events in-memory and over a short time window. However, users often want to perform longer-term analyses over large time windows on the data streams, e.g. to determine the conditions that should raise alerts.

While it is possible to build separate systems for either real-time or long-term data analysis, a system which provides both capabilities is more useful. The window of data used for queries can seamlessly range from short term to very long term, making it difficult to decide where to divide the systems. Furthermore, historical data can provide a context for interpreting new data [2]. A *stream warehouse* bridges the short-term vs. long-term gap by loading data continuously in a streaming fashion and warehousing them over a long time period (e.g. years). Stream warehouse systems, such as Moirae [2], latte [22], DataDepot [11], Everest [1], and Truviso [10], have been applied to monitoring applications such as data centers [2], RFID [23], web complexes [1], highway traffic [22], and wide-scale networks [14].

A DSMS normally monitors a nearly-instantaneous and ordered data feed of, e.g., network packets [8], financial tickers or sensor measurements. However, a stream warehouse operates on longer time scales, and, instead of processing data from a localized source, it receives a wide range of data feeds from disparate, far-flung, and uncontrolled sources. For example, the Darkstar network management system [14] (built using DataDepot) receives more than 100 distinct data feeds, each of which collects data from a worldwide communications network using many different dissemination mechanisms. These distinct feeds need to be cross-correlated and analyzed into higher level data products for use by network analysts. In such a widely distributed and heterogeneous environment, one can no longer assume that data within a stream arrive in time-order (or nearly so), or that streams are synchronized with each other. This leads to new *temporal consistency* problems: we want to load new data (and propagate changes to the materialized views maintained by the warehouse) as quickly as possible, but "stable" results may require significant synchronization delays. (Note that the temporal consistency issues studied in this paper are orthogonal to transactional consistency issues that arise from multiple data writers and/or readers.)

Consider a network monitoring system that collects performance measurements, such as router CPU utilization or the number of packets forwarded, and various system logs. Suppose that an alerting application generates an alarm whenever the CPU usage of a router exceeds a supplied threshold. If a high-CPU-usage record arrives, the application should not have to wait until all temporally preceding data have arrived before taking action. Similarly, a view containing all the routers that have generated at least ten critical system log messages in any one-minute window can be updated whenever the message count for a particular router, call it *r*, reaches ten; we do not need to see data from other routers, nor do we need to wait and see if any more messages from *r* arrive in this window. On the other hand, suppose that we want to maintain aggregated statistics for each time window. It may be better to wait until all the expected measurements have arrived before updating the statistics over the latest window, both in terms of interpretability (aggregates computed on incomplete data may not be accurate) and update efficiency (we want to avoid re-computing expensive aggregates while data are still trickling in).

These types of problems become even more challenging in production stream warehouses that correlate a wide variety of highly disordered and asynchronous feeds and maintain complex

5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

materialized view hierarchies. Such warehouses often support critical applications; examples from the networking domain include real-time network troubleshooting and anomaly detection [14]. However, without an understanding of temporal data consistency, we may not know how to trust the answers.

Motivated by our experiences with production stream warehouses, we present *temporal consistency* models for a stream warehouse that range from very weak to very strong, and we show how they can be tracked and used simultaneously. Given that warehouse tables are typically partitioned by time, the key technical novelty is to reason about and to propagate consistency information at the granularity of partitions. Since a significant part of the value of a stream warehouse is its ability to correlate disparate data sources for the users, our models describe the state of the data in an intuitive way that allows users to interpret real-time query results. For instance, a partition that is guaranteed not to change is marked "closed", while one that may be updated with new data, but whose existing data are guaranteed not to change, is marked "append-only". Since warehouse maintenance involves propagating changes across view hierarchies, we also discuss disseminating consistency level information from base tables to materialized views and vice versa. We show that stronger consistency levels not only provide assurances for query results, but they can also be used to avoid unnecessary computations. Finally, we discuss applications of our models to monitoring *data stream quality*.

2. BACKGROUND AND MOTIVATION

A DSMS continuously ingests data from one or more *data feeds*, and processes a collection of long-running queries over these feeds. Many sources can produce a data feed: a stream of measurements, log files delivered from an external source, a log of updates to a transactional store, and so on. The feed regularly presents a *package* of records for ingest into the stream system. The records in a package are stamped with the time of the observation (or observation time interval), and also the package itself is often timestamped. The set of timestamps in a package are generally highly correlated with the package timestamp and delivery time.

Data feeds are usually append-only; i.e., records that have arrived in the past are not deleted or modified in the future. For example, a feed of network measurements may have a schema of the form *(timestamp, router_id, avg_cpu_usage)*, with each record corresponding to the average CPU usage of the router with the given *router_id* recorded at the given time(stamp). We may receive a new package every five minutes, containing new CPU usage measurements for each router. Here, data from old packages (old measurements) are never deleted or modified. However, in some applications, old packages may be revised and retransmitted.

When a package arrives in a DSMS, the conventional behavior is to fully process the new records (modulo operator scheduling policies [5]). Some exceptions occur: a *sort* operator might reorder slightly disordered streams, and blocking operators such as aggregation and outer join might delay some or all of their output until a punctuation [21] indicates end-of-window. However, these mechanisms assume that streams are mostly-synchronized and mostly-ordered, so that buffering costs and processing delay times are small (the discussion of punctuation generation in [13] implicitly assumes that streams are synchronized).

As mentioned, a stream warehouse faces more challenging problems of disorder in its input streams. We have found the following disorder problems within the Darkstar warehouse:

Data arrive in a smear over time

In the course of operating several DataDepot warehouses, we noticed that any given package of data contains records with a range of timestamps. This behavior is not unexpected since data are gathered from world-wide network elements. We investigated this phenomenon by examining the data arrivals of several Darkstar tables.

We first examined arrivals for table C, which contains 5-minute statistics about router performance – a package normally arrives once every 5 minutes. We found that 23 percent of the packages (covering a 10-day period) contain some data for a previous 5-minute period, and sometimes for data up to an hour old (the packages frequently arrive late also). In another table, T, loaded at 1-minute intervals, every package except one contained records for a previous time period (observed over a 7-day interval). A third table, S (loaded at 1-minute intervals), showed the greatest disorder: each package contained data for an average of 4.5 previous time periods. The degree of disorder changes over time, as illustrated in Figure 1 which plots the number of time periods with at least one record in any given package. We hypothesize that the degree of disorder within a package is related to load on the data delivery system.

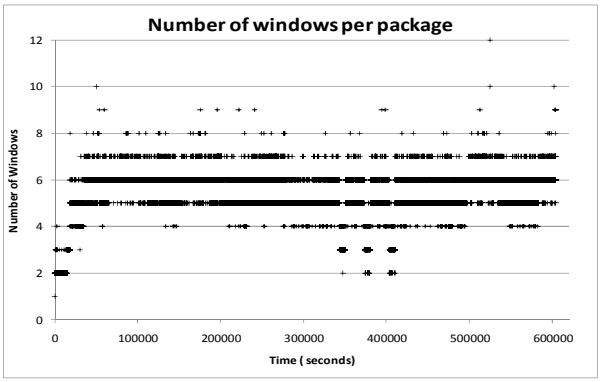


Figure 1. Number of time periods in one package for S

Data sources are unsynchronized

Different data feeds use different collection and delivery mechanisms, and therefore they tend to have different degrees of currency. We considered three feeds, the previously mentioned C and T (containing router alerts), and a third feed WD (packet loss and delay measurements), and sampled the lateness of the most recent data in each of these tables. On average, T was 6 minutes behind, C was 17 minutes behind, and WD was 47 minutes behind. Again, we believe that the currency of these feeds changes according to the load on their data delivery system.

Late arrivals are common

Significantly late arrivals are not common enough to readily measure, but in our experience they occur often enough to be an

operational concern – corroborated by another recent study [15]. Often the problem is a temporary failure of a component in the data delivery system. Occasionally, a portion of the source data is discovered to be corrupt and needs re-acquisition and reloading.

Given the large data volumes and high disorder in the source streams of a stream warehouse, conventional in-memory buffering techniques are prohibitively expensive [10]. Compounding the problem are complex view hierarchies. For example, Figure 2 shows a fragment of a real-time network monitoring application which searches for misbehaving routers, involving WD and other data (the full application has another 21 tables). The octagons are the base tables, while boxes identify tables that are often queried. These types of applications are too large to manage using conventional means and too complex to be understood without consistency assurances.

Another problem is that there can be multiple notions of consistency that users desire. For example, some Darkstar users (or applications) require access to router alerts (e.g., T) as soon as possible, and need to correlate them with the most recent possible router performance reports (e.g., C). Other users (or other materialized views) might need stable answers to queries based on these streams, even at the cost of a moderate synchronization delay.

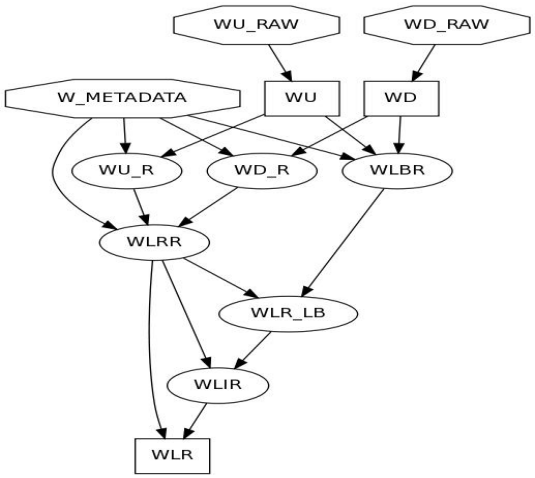


Figure 2. Data flow in an application fragment

3. SYSTEM MODEL

This work was motivated by the practical problems encountered by users of our DataDepot stream warehouse. We phrase the system model in DataDepot terms, but the model applies to all of the stream warehouses we have seen (perhaps with a change of phrasing).

A stream warehouse is characterized by streaming inputs, by a strong emphasis on the temporal nature of the data, and by multiple levels of materialized views. To manage a long-term store of a data stream, the stream is split into temporal *partitions* (or panes [16], windows [4][10], etc.). Each temporal partition stores data within a contiguous time range. The collection of temporal partitions of a stored stream comprises a complete and non-overlapping range of the stored window of the data stream. A feed package may contain data for multiple partitions, as shown in Figure 1. The storage of a high-volume stream may require additional partitioning dimensions, but we will not be concerned with this complication in this paper.

A data warehouse maintains a collection of *materialized views* computed from the raw inputs to the warehouse. Materialized views are used to accelerate user queries by pre-computing their answers and to simplify data access by cleaning and de-normalizing tables. A stream warehouse typically has a large collection of materialized views arranged as a Directed Acyclic Graph (DAG). The DAG tracks data dependencies, e.g. that view V is computed from streams A and B (Figure 2 shows a data flow DAG, the reverse of a dependency DAG). A stream warehouse also tracks temporal dependencies, e.g. that data in V from 1:00 to 1:15 are computed from data in stream A from 1:00 to 1:15 and from data in B from 12:30 to 1:15 (as in Figure 3).

Let V be a warehouse table. We assume that V has a *timestamp field*, **V.ts** which tends to increase over time. Further, we assume that every table V is temporally partitioned, and that the partitions are identified by integer values so that **V(t)** is the t^{th} partition of V. Associated with V is a strictly increasing *partitioning function* $\boldsymbol{pt_V(t)}$. Partition *t* of V contains all and only those data in V such that

$$pt_V(t) \leq V.ts < pt_V(t+1).$$

Base tables are loaded directly from a source stream (for example, WU_RAW in Figure 2). Derived tables (materialized views) are defined by a query over other base and derived tables (for example, WU_R in Figure 2). We define **S(V)** to be the set of source tables of V, e.g. S(WU_R) = {WU, W_METADATA}. We assume that all derived-table-defining queries exhibit temporal locality (e.g., they may be defined over a sliding window).

Let S be a table in S(V). Then **Dep(V(t),S)** is the set of partitions in S that supply data to V(t), and **Dep(V(t))** is the set of all partitions that supply data to V(t) regardless of the source table. For example, suppose that in Figure 3, each partition represents 15 minutes of data, and that partition 20 corresponds to 1:00 through 1:15. Then Dep(V(20), B) = {B(20)} and Dep(V(20)) = {A(18), A(19), A(20), B(20)}. When any of the partitions in Dep(V(20)) are updated, V(20) must also be updated (incrementally, if possible, or by being re-computed from scratch).

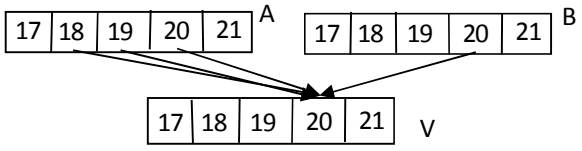


Figure 3. Partition dependencies

4. CONSISTENCY MODELS

Our basic notion of temporal consistency assigns one or more markers to each temporal partition in a table. Consistency markers can be thought of as a generalization of punctuations, since multiple consistency levels would be used in general. Below, we propose two related but different notions of

consistency. The first, *query consistency* defines properties of data in a partition that determine if those data can be used to answer a query with a desired consistency level. The second, *update consistency*, propagates table consistency requirements and is used to optimize the processing of updates to a stream warehouse.

4.1 Query Consistency

Our definition of query consistency starts at the base tables. For the purposes of this discussion, we use a minimal set of three levels of consistency, but many more are desirable in practice. We choose this particular set of three levels because they are natural and they form a simple hierarchy, but they also illustrate some interesting aspects of query consistency. However, an actual implementation of a warehouse would likely use a more refined set of consistency levels, as we will discuss in Section 5.

Let B be a base table and let B(d) be one of its partitions. Then:

- **Open(B(d))** if data exist or might exist in B(d).
- **Closed(B(d))** if we do not expect any more updates to B(d) according to a supplied definition of expectation; e.g., that data can be at most 15 minutes late.
- **Complete(B(d))** if Closed(B(d)) and all expected data have arrived (i.e., no data are permanently lost).

The notions of Open and Closed consistency are the natural minimal and maximal definitions. Complete consistency is stronger, and it is motivated by DataDepot user requirements: only perform analysis on complete data partitions because otherwise one may get misleading results (however, Closed partitions are often acceptable to users). Of course, the vagaries of the raw data sources may make it difficult to precisely establish when a partition has achieved one of these levels of consistency; this is similar to the problem of generating punctuations. However, several types of inference are possible:

- If there is at least one record in a partition, we mark it as Open. However, a partition might have Open consistency even though it is empty: no data might ever be generated for it. We might mark an empty base table partition as Open if we can infer that some data could have arrived, e.g. if a temporally later partition is non-empty.
- We might know that exactly five packages provide data for a partition and that packages rarely arrive more than one hour late. If so, we can mark a partition as both Closed and Complete if all five packages have arrived. If only four have arrived, but an hour has passed since the expected arrival time of the fifth one, we would only mark the partition as Closed. If the fifth package never arrives, this partition never becomes Complete.

The consistency of a partition of a derived table is determined by the consistency of its source partitions. Each level of consistency has its own inference rules, and inference is performed for each consistency level separately. The most basic inference rule is as follows: *for consistency level C, infer C(V(t)) if C(S(d)) for each S(d) in Dep(V(t))*. However, by analyzing the query that defines a materialized view we can sometimes create a more accurate inference rule.

Let us consider an example set of inference rules using our set of three consistency levels. Let V be a derived table and let V(t) be one of its partitions.

Query Consistency Inference

- Let RQD(V), a subset of S(V), be the non-empty set of tables referenced by "required" range variables, i.e., those used for inner-join or intersection.
  - If RQD(V) is non-empty, then **Open(V(t))** if for each S in RQD(V), there is a S(d) in Dep(V(t),S) such that Open(S(d)).
  - If RQD(V) is empty, then **Open(V(t))** if there is a S(d) in Dep(V(t)) such that Open(S(d)).
- **Closed(V(t))** if Closed(S(d)) for each S(d) in Dep(V(t)).
- **Complete(V(t))** if Complete(S(d)) for each S(d) in Dep(V(t)).

The Closed and Complete consistency levels use the basic inference rule, but by analyzing the query that defines materialized view V we can avoid labeling a partition V(t) as Open when no data can be in it. Section 5 contains additional examples of query-dependent consistency inference rules.

The inference that a partition of a derived table has a particular consistency level is computed top-down (from source to dependent tables). Normally, this inference would be performed at view maintenance time by comparing source with destination consistency metadata. This maintenance can be performed globally, as with, e.g., Oracle [9], or piecemeal, as with DataDepot [11]. Note that the consistency of a partition can change even though the partition does not need to be updated, e.g., due to a base table partition becoming Closed as well as Open.

For example, consider table V computed by an inner join of A and B as shown in Figure 4. In this figure, we represent Open, Closed, and Complete consistency markers by O, Cl, and CM, respectively, and we omit an O marker if a Cl marker exists. Partition 1 of V can be inferred to have Closed consistency, since both sources are Closed, but not Complete consistency; however partition 2 can be inferred to be Complete. Partition 3 is Open because both A and B can contribute an Open (or Closed) partition. Partition 4 cannot even be inferred to be Open.

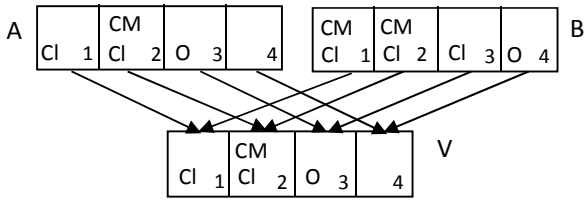


Figure 4. Query consistency inference

Query consistency markers ensure the consistency of query results. In Darkstar applications, ensuring temporal consistency is critical, but very difficult without warehouse support. Applications such as RouterMiner and G-RCA [14] enable real-time network troubleshooting by correlating data from feeds including C, S, T, WD and many others; however, each of these

feeds produces base tables with widely varying timeliness (recall Section 2).

An outline of the procedure for ensuring the consistency of a query is to treat the query as a derived table and determine its partition dependencies. A query can be answered with a given level of consistency if that consistency level can be inferred from the set of all source partitions accessed by the query. A query that cannot be answered with the desired consistency can have its temporal range trimmed (or its consistency relaxed). For example, if we are performing a selection on table V in Figure 4 and we require Complete consistency, then the inference rules state that the query can only be run on the data in partition 2.

While the proposed mechanism for ensuring query consistency is general, it can be confusing to users. A convenient way to summarize the state of a (base or derived) table is a *consistency line*. The C-consistency line of table V is the maximum value of *pt* such that all partitions V(t), $t \leq pt$, have C(V(t)). A query that references tables S_1 through S_n can be answered with C-consistency if it is restricted to accessing partitions of S_i at or below the C-consistency line of S_i for each i=1,..,n. In previous literature, we have referred to the Open-consistency line as the *leading edge* of a table, and the Closed-consistency line as the *trailing edge* [11]. A Complete-consistency line is likely of little value since some partitions might permanently fail to become Complete.

For example, the Open-line (leading edge) of table V in Figure 4 is partition 3, while the Closed-line (trailing edge) of V is partition 2. We cannot define a Complete line since partition 1 is not Complete.

4.1.1 Case Study

We now give an example of how applications can choose and exploit query consistency guarantees. A fragment of one of the Darkstar applications was shown in Figure 2. This application processes packet delay and packet loss measurements to come up with network alarm events. These measurements are taken roughly every five minutes, one measurement for each link in the network. A loss or delay alarm record is produced for a given link if there are four or more consecutive loss or delay measurements, respectively, that exceed a specified threshold. If a measurement for a given link is missing in a 5-minute window, it is considered to have exceeded the threshold for the purposes of alarm generation. In Figure 2, WLR is the materialized view that contains loss alarm records, each record containing a link id, the start and end times of the alarm, and the average packet loss and delay during the alarm interval. The size of each WLR partition is five minutes, which corresponds to the frequency of the underlying data feeds. The ovals in Figure 2 correspond to intermediate views that implement the application logic (e.g., selecting measurements that exceed the threshold, computing the starting point of each alarm event, computing alarm statistics, etc.). To complete the application, a Web-based front end displays the current and historical alarms by periodically querying the WLR table.

Since this is a real-time alerting application, one may argue that WLR should have Open consistency; i.e., it should be loaded with all the available data at all times. However, the problem is that missing measurements are assumed to have exceeded the threshold. Thus, if we attempt to update WLR before the latest measurements arrive, we will incorrectly assume that all of these measurements are missing and we may generate false alarms. Instead, it is more appropriate to use Closed consistency for WLR, with partitions closing at each 5-minute boundary. Note that Complete consistency may not be appropriate for this application since we do not want to delay the generation of network alarms for the data that have already arrived, even if a partition is not yet complete.

4.2 Update Consistency

In addition to understanding data semantics and query results, another use for consistency is to minimize the number of base table and view updates in a warehouse. For an example drawn from experience, consider a derived table V defined by an aggregation query which summarizes a real-time table S with once-per-5-minutes updates (with 5-minute partitions) into a daily grand-total summary (with per-day partitions). If V is updated every time S is updated, V would be updated about 288 times (1440 minutes in a day / 5) before the day is closed out. If we are interested in the grand total rather than the running sum, this procedure for updating V is wasteful. Here, a partition of V is only useful if it has Closed consistency, so we should only compute it when it can be safely Closed.

The *update consistency* of a table is the minimal consistency required by queries on it or its dependent tables, and determines when to refresh its partition(s). A partition of a table is computed only when it can be inferred to have a query consistency matching the desired update consistency.

Naively, we might require the warehouse administrator to mark each table with its desired update consistency. However, any given table may supply data to many derived tables, each with differing types of update consistency. We need an algorithm for determining what kind of update consistency table S should enforce.

Furthermore, not every view is primarily intended for output. A table might be materialized to simplify or accelerate the materialization of another table, or it might be a partial result shared by several tables (see, e.g., the application fragment in Figure 2). We assume that output tables are marked as such (all leaf-level materialized views are output tables). A table can be marked with one of the following labels:

- **Prefer_Open**: a table that does not have to reflect the most recent data, but one whose partitions can be easily updated (in an incremental manner) if necessary; e.g., *monotonic* views such as selections and transformations of one other table.
- **Require_Open**: a real-time table in which any possible data must be provided as soon as possible.
- **Prefer_Closed**: Tables whose partitions are expensive to re-compute, such as joins and complex aggregation (depending on the incremental maintenance strategy).
- **Prefer_Complete**: a table whose output is only meaningful if the input is complete.

All output tables need to be marked with these initial labels, which may be more effort than the warehouse administrator cares to expend. By default, selection and union views may be marked Prefer_Open because they can be very easily updated. Join and

aggregation views may be marked Prefer_Closed since it is more efficient to perform batch updates to them rather than continuously updating them whenever new data are available (or because users may not be interested in partial aggregates). We note that Prefer_Open is a "don't care" type of condition.

The algorithm for determining update consistency works in a reverse breadth-first search (BFS) of the data flow DAG, starting from the leaf-level views and working to the roots (base tables). When a table T is selected for processing, all of its dependent tables have received their final marking. To mark table T, we follow a resolution procedure. Let M be the set of dependent table markings, along with the marking of table T, if any (internal-use tables might not be marked).

The three consistency levels we are using form a hierarchy: Complete implies Closed, and Closed implies Open. The general resolution procedure is to choose the lowest level of consistency in M, with the "don't care" consistency level as a fallback. Therefore our update consistency resolution procedure is simple and produces a single result. There is one complication: it is likely that not all base table partitions will ever be labeled Complete, and therefore we should use Complete update consistency only if all dependent tables use Complete update consistency.

Update Consistency Resolution:

1. If Require_Open is in M, mark T as Require_Open
2. Else, if some label in M is Prefer_Closed, mark T as Prefer_Closed
3. Else, if all labels in M are Prefer_Complete, mark T as Prefer_Complete
4. Else, mark T as Prefer_Open.

Tables marked Require_Open or Prefer_Open use Open update consistency, while tables marked Prefer_Closed (resp. Prefer_Complete) use Closed (resp. Complete) update consistency.

Consider the example illustrated in Figure 5. The leaf tables (V, W, X, Y) are all output tables, indicated by a rectangle, with pre-assigned update consistency levels of (Require_Open, Prefer_Complete, Prefer_Closed, Prefer_Open) respectively. These tables are considered first in the reverse BFS search of the DAG. When one of these tables is processed, its own label is the only entry in M, so each table in (V, W, X, Y) is assigned its preferred update consistency. Non-output tables (A, B, C) are processed next. When one of these tables is processed, the markings of its successor tables are the entries in M. For example, when B is processed the entries in M are (Prefer_Complete, Prefer_Closed) so the resolution procedure marks B as Prefer_Closed.

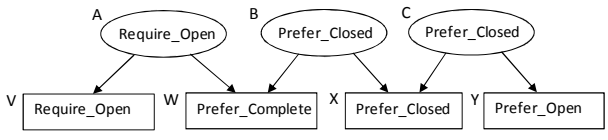


Figure 5. Update consistency inference

4.2.1 Experimental Evaluation

To see the potential performance benefit of using update consistency, we collected the number of updates performed on tables WU_RAW, WD_RAW, and WLR in Figure 1, over a 10 day, 18+ hour period. WU_RAW and WD_RAW (are supposed to) receive updates every 15 minutes; therefore we expect 1033 updates to these tables during the observation period. W_METADATA is another input, but it receives updates only once per day, so we will ignore it in our analysis.

The implementation of DataDepot that we measured did not incorporate update consistency. The only update scheduling options available were *immediate* (update a table whenever one of its sources has been updated) and *periodic* (e.g. every 15 minutes). Since the W application is real-time critical, we used immediate scheduling to minimize data latency. The extensive use of joins in this application suggests that immediate updates are likely to be inefficient, since one will often perform an (inner) join update when data from only one range variable is available. Without an update consistency analysis, however, the warehouse has no basis for *not* performing an update when data from only one range variable is available, since the join might be an outer join, and which source table supplies the outer join range variable is not clear.

During the observation period, there were 1364 updates to WU_RAW and 1359 to WD_RAW. The excess over the expected 1033 updates are due to late arrivals of some of the packages that comprise the data in a partition (recall the discussion in Section 2). Under Open update consistency, the number of updates to WLR should be 1033 plus one for each excess update to one of the RAW tables, for a total of 1690 updates. We observed 4702 updates to WLR: 3012 unnecessary updates. The use of update consistency clearly has the potential to be a significant optimization since we could reduce the number of updates to WLR by 64 percent. If we used Closed consistency, we could reduce the number of updates by 78 percent.

5. EXTENSIONS

Our framework for computing and using query and update consistency is general and can be extended to additional consistency levels, as long as one can produce safe inference rules.

As we discussed in Section 4, determining the consistency level of a base table partition is a matter of guesswork. Closed partitions are generally not really closed since very late data might arrive, sources might provide revisions to previously loaded data ("Sorry, we sent you garbage"), and so on. Thus, it may be useful to provide different levels of closed-ness according to different definitions. For example, WeakClosed(V(t)) might mean that the data are probably loaded and stable enough for queries, while StrongClosed(V(t)) might mean that we are certain enough that no more data will arrive and that we will refuse to process revisions. The definition of Closed in Section 4.1 corresponds to WeakClosed here.

If we have reasonably accurate and stable statistics about late arrivals, we can associate specific time-out values with various levels of closed-ness. For instance, we may know that revisions and updates mostly occur within five minutes of the expected partition closing time, and very few occur an hour later. A similar

example is X-Percent-Closed(V(t)), which indicates that X percent of the data will not change in the future. This consistency marker is motivated by nearly-append-only data feeds that we have observed in the Darkstar warehouse, which are mostly stable except for occasional revisions. Finally, different levels of completeness, such as X-Percent-Full(V(t)) may also be useful --- the warehouse may maintain summary views whose results are acceptable as long as they summarize a sufficient fraction of the input.

The above types of consistency levels may be used to quantify and monitor data quality in a stream warehouse. For example, if the number of packages per partition is a fixed constant, we can track multiple X-Percent Closed and Full consistency lines for various values of X in order to understand the extent of missing and delayed data. Such consistency lines may also be useful for monitoring and debugging the warehouse update propagation algorithm. For example, if all the base tables are full, but recent derived table partitions are only 25 percent full, or just open, then perhaps the warehouse is spending too much time trying to keep up with the raw inputs rather than propagating updates through materialized views. We hope to report on a visual data quality tool based on consistency lines in future work.

Conventionally defined punctuations allow for group-wise processing, i.e., an assurance that all data within a group have arrived. Analogously, in some cases we might be able to provide group-wise consistency guarantees if we know that, e.g., all data from routers in the European region has arrived while we are still waiting for data from Southeast Asia routers. Propagating group-wise punctuation would require a more sophisticated analysis of the queries that define materialized views, e.g. that an aggregation query groups on the region column.

If we are willing to make increasingly detailed analyses of the queries that define tables, we can obtain a more refined and less restrictive set of consistency levels. Three additional types of consistency are:

- **NoNewRecords**: no records will be added to the partition in the future, but some existing records may be removed (this may happen in views with negation).
- **NoFieldChange**(K,F): If a record with key K exists in the partition, the value of fields K union F will not change in the future.
- **NoDeletedRecords**: no records will be deleted from this partition in the future, but new records may be added (this occurs in monotonic views).

A full description of how to analyze queries to apply these consistency levels is lengthy and detailed. However, we outline one type of query as an example. Suppose that table V is computed by outer-joining B to A, and the join predicate is from a foreign key on A to a primary key on B. Then NoNewRecords(V) depends on NoNewRecords(A) and NoFieldChange(A) only, not on table B.

5.1 Update Consistency in the Presence of Multiple Hierarchies

The discussion of update consistency resolution in Section 4.2 assumes that the collection of consistency levels form a hierarchy, e.g., Complete(V(t)) => Closed(V(t)) => Open(V(t)). However, a complex collection of consistency levels is likely to have many incomparable definitions. For example, from WeakClosed(V(t)) we cannot infer 100-Percent-Full(V(t)), nor vice versa. In this section we show how to resolve the update consistency of a table in a general setting.

Let **Cn** be the set of consistency classes available to the warehouse. We define a predicate **Stronger(C1,C2)**, C1 and C2 in Cn, if C1(V(t)) => C2(V(t)). We assume that the pair (Cn, Stronger) forms a directed acyclic graph, Gc, that includes all transitive edges.

Consistency classes such as Closed, in which some partitions might never reach the specified level of consistency, lead us to make additional definitions. For consistency level C in Cn, we define **Linear(C)** if C(V(t)) => C(V(t-k)) for $0 < k < t$. We also choose a default consistency level, C*default* in Cn, to be the update consistency level to be used if the update consistency resolution procedure returns an empty result.

As in Section 4.2, let M be the set of dependent table markings, along with the marking of table V. Let **Dep** the set of children of T in the data flow DAG. Then:

Update Consistency Resolution with hierarchies

1. Mark a node C in Cn if
   a. Linear(C), and C in M.
   b. Not Linear(C), C in M, and for each D in Dep, the update consistency of D is C.
2. Let U be the marked nodes {C1} in Cn such that there is no C2 in Cn such that Stronger(C1,C2).
3. If U is non-empty
   a. Return U
   b. Else return {C*default*}

Let U(V) be the set of update consistency levels returned by the update consistency resolution procedure. Then a partition V(t) is updated if we can infer consistency level Cu(V(T)) for some Cu in U(V).

We now present examples to illustrate update consistency inference with multiple hierarchies. Suppose that Cn={C1,…,C7}, and neither C5 nor C7 are linear (which we note with the double lined circle). Edges imply the Stronger relation (e.g. Stronger(C5,C3)), and we have removed transitive edges for clarity. In Figure 6, M = {C2, C3, C6}, so the result is that U = {C3, C3}. In Figure 7, M = {C5, C4}. However, not every S in Dep has update consistency C5 (as witnessed by the C4 marking}, and therefore we do not use C5 in the resolution procedure. Therefore U = {C4}.

6. RELATED WORK

The type of consistency we discuss in this paper relates to *temporal consistency* rather than transactional consistency. Data warehouses often use locking [9] or multi-version concurrency control [11][19] for the latter. However the method for implementing transactional consistency is orthogonal to the concerns of this paper.

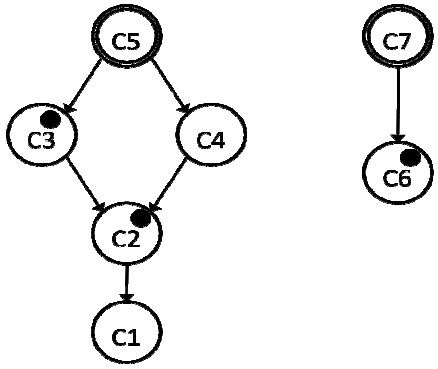


Figure 6. Update consistency resolution (a)

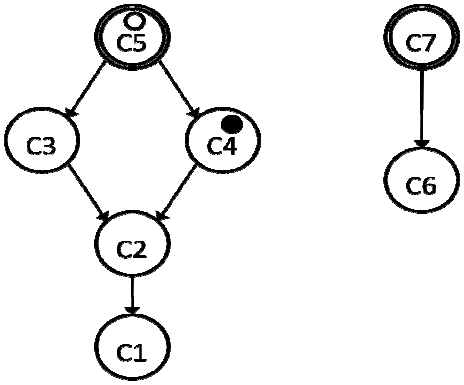


Figure 7. Update consistency resolution (b)

Materialized view maintenance in a data warehouse has an extensive literature; we summarize some key points below. The notion of temporal consistency in a data warehouse is generally taken to mean some type of *strong* consistency [25], e.g., that all materialized views are sourced from the same data, generally meaning that all views are updated in a single global pass. While some work has been done to allow for multiple consistency zones [6][24] using different consistency policies (e.g., immediate vs. deferred updates), any table belongs to a single zone and all tables in a zone are updated together. Even modern data warehousing systems are oriented towards batch updates [9]. The Real-Time community often defines a database as being consistent if it contains data representing a recent time interval, and mutually consistent if tables represent the same time intervals [12].

Temporal databases often use the *bitemporal model* [20]. Each record in a bitemporal database has a *valid time*, which refers to the time interval during which an event occurred, and a *transaction time*, which refers to the system clock time when a record is the most recent description of an event. However, the bitemporal model is not useful for much of the data in a stream warehouse (temporal metadata tables are a notable exception): the analyst is not concerned about transaction time, records in event feeds generally have many often conflicting timestamps and sequence numbers, and bitemporal databases do not enable update consistency.

Conventional DSMSs usually assume that data are in-order or nearly so, and they manage disorder by sorting or by punctuations and a limited degree of "out-of-order" processing [17]. Query operators can be assumed to have the most recent data, and consistency becomes a non-issue (but see [3], which manages stream consistency using revisions). We refer the interested reader to the more detailed discussion in Section 2 of [15].

Concepts similar to Closed consistency were discussed in [3], but that work assumed a specific tri-temporal data model and focused on handling revisions. Our consistency models only assume that each record has a timestamp whose value tends to increase over time, and they are oriented towards users' trust in query answers and efficient warehouse maintenance.

Another interesting comparison is between the two stream warehouse systems whose consistency management has been most fully described: DataDepot (in this paper and in [11]), and Truviso [10][15]. These two systems have approached stream warehousing from different angles: DataDepot adds stream processing to a conventional data warehouse, while Truviso adds warehousing capabilities to a stream system.

Truviso allows stream queries to reference conventional database tables, which can be updated during stream processing. Truviso uses *window consistency* [7] for these types of queries: the processing of stream S on window w has read-consistency on table T during the processing of w. To handle late-arriving data, Truviso computes window revisions [15], which can be thought of as the increments for self-maintaining views [18].

The consistency mechanisms described in this paper and those described for Truviso are orthogonal. DataDepot could benefit from window consistency (currently it uses temporal metadata tables such as W_METADATA in Figure 2). The window revisions are an optimized method for performing view maintenance, as compared to DataDepot's default of recomputing partitions affected by new data (Truviso falls back to recomputing affected windows for views that are non self-maintaining [15]). Thus, the consistency mechanism described in this paper applies to both systems.

7. CONCLUSIONS AND FUTURE WORK

We proposed mechanisms for managing and exploiting the consistency of materialized views in a stream warehouse. The first, *query consistency*, propagates consistency properties from base tables to materialized views, and provides consistency guarantees of query results. The second, *update consistency*, propagates table consistency requirements from materialized views to base tables, and is used to optimize the management of a stream warehouse. We focused on a most basic three types of consistency: Open, Closed, and Complete; however, as discussed in Section 5, many more useful consistency definitions can fit within our framework.

There are several issues not fully addressed in this paper. One issue is the handling of very late data, e.g. data that arrive long after the base table partitions have been marked Closed. These partitions, and all dependent partitions in dependent tables, need to be recomputed, but what is the best way to handle the revisions to the consistency markings? We have proposed the "trailing-edge line" as a convenient way to summarize stable data, but late arrivals poke holes in this line.

Another issue which is not fully addressed is the processing of query-specific consistency properties. Our basic models make

some use of query-specific handling, e.g. inner-join vs. outer-join range variables. However, query-specific consistency inference can become arbitrarily complex; experience will determine whether the complexity produces a tangible benefit.

All the examples in this paper assumed that recent data may be suspect, but they eventually stabilize over time. We are also interested in applying our consistency framework to data that begin as "exact" when loaded into the warehouse and then "decay" or lose accuracy over time. Examples include location data periodically collected from moving objects, and sensor measurements.

8. REFERENCES

[1] M. Ahuja, C. C. Chen, R.Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala, Peta-scale data warehousing at Yahoo!, *Proc. of SIGMOD 2009*, 855-862.

[2] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee, Moirae: History-Enhanced Monitoring, *Proc. of CIDR 2007*, 275-286.

[3] R. Barga, J. Goldstein, M. Ali, and M. Hong, Consistent Streaming Through Time: A Vision for Event Stream Processing. *Proc. of CIDR 2007*, 363-374.

[4] I. Botan, R. Derekhshan, N. Dindar, L. Haas, R. Miller, and N. Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB* 3(1):232-243 (2010).

[5] R. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, Operator Scheduling in a Data Stream Manager, *Proc. of VLDB 2003*, 838-849.

[6] L. Colby, A. Kawaguchi, D. Lieuwen, I. S. Mumick, and K. Ross, Supporting Multiple View Maintenance Policies, *Proc. of SIGMOD 1997*, 405-416.

[7] N. Conway, Transactions and Data Stream Processing, http://neilconway.org/docs/thesis/pdf, April 2008.

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, Gigascope: A Stream Database for Network Applications, *Proc. of SIGMOD 2003*, 647-651.

[9] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng, Optimizing Refresh of a Set of Materialized Views, *Proc. of VLDB 2005*, 1043-1054.

[10] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, Continuous Analytics: Rethinking Query Processing in a Network-Effect World, *Proc. of CIDR 2009*.

[11] L. Golab, T. Johnson, J. Seidel, and V. Shkapenyuk, Stream warehousing with DataDepot, *Proc. of SIGMOD 2009*, 847-854.

[12] A.K. Jha, M. Xiong, and K. Ramamritham. Mutual Consistency in Real-Time Databases. *Proc. of the 27th IEEE Real Time Systems Symposium (RTSS) 2006*, 335-343.

[13] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck, A Heartbeat Mechanism and Its Application in Gigascope, *Proc. of VLDB 2005*, 1079-1088.

[14] C. Kalmanek, Z. Ge, S. Lee, C. Lund, D. Pei, J. S. Seidel, K. Van der Merwe, and J. Yates, Darkstar: Using Exploratory Data Mining to Raise the Bar on Network Reliability and Performance, *Proc. of DRCN 2009*.

[15] S. Krishnamurthy, M. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre, Continuous analytics over discontinuous streams, *Proc. of SIGMOD 2010*, 1081-1092.

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker, No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record* 34(1): 39-44 (2005).

[17] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, Out-of-order processing: a new architecture for high-performance stream systems, *PVLDB* 1(1): 274-288 (2008).

[18] I. Mumick, D. Quass, and B. Mumick, Maintenance of Data Cubes and Summary Tables in a Warehouse, *Proc. of SIGMOD 1997*, 100-111.

[19] D. Quass and J. Widom. On-line warehouse view maintenance. *Proc. of SIGMOD 1997*, 393-404.

[20] R.T. Snodgrass. The TSQL2 Temporal Query Language, Kluwer 1995.

[21] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, Exploiting Punctuation Semantics in Continuous Data Streams, *TKDE* 15(3): 555-568 (2003).

[22] K. Tufte, J. Li, D. Maier, V. Papadimos, R. Bertini, and J. Rucker, Travel time estimation using NiagaraST and latte, *Proc. of SIGMOD 2007*, 1091-1093.

[23] E. Welbourne, N. Khoussainova, J. Letchner, Y. Li, M. Balazinska, G. Borriello, and D. Suciu, Cascadia: a system for specifying, detecting, and managing RFID events, *Proc. of MobiSys 2008*, 281-294.

[24] Y. Zhuge, H. Garcia-Molina, and J. Wiener, Multiple View Consistency for Data Warehousing, *Proc. of ICDE 1997*, 289-300.

[25] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, View Maintenance in a Warehousing Environment, *Proc. of SIGMOD 1995*: 316-327.

Deuteronomy: Transaction Support for Cloud Data

Justin J. Levandoski[1§] David Lomet[2] Mohamed F. Mokbel[1§] Kevin Keliang Zhao[3§]

[1]University of Minnesota, Minneapolis, MN, {justin,mokbel}@cs.umn.edu
[2]Microsoft Research, Redmond, WA, lomet@microsoft.com
[3]University of California San Diego, La Jolla, CA, kezhao@cs.ucsd.edu

ABSTRACT

The Deuteronomy system supports efficient and scalable ACID transactions in the cloud by decomposing functions of a database storage engine kernel into: (a) a transactional component (TC) that manages transactions and their "logical" concurrency control and undo/redo recovery, but knows nothing about physical data location and (b) a data component (DC) that maintains a data cache and uses access methods to support a record-oriented interface with atomic operations, but knows nothing about transactions. The Deuteronomy TC can be applied to data *anywhere* (in the cloud, local, etc.) with a variety of deployments for both the TC and DC. In this paper, we describe the architecture of our TC, and the considerations that led to it. Preliminary experiments using an adapted TPC-W workload show good performance supporting ACID transactions for a wide range of DC latencies.

1. INTRODUCTION

The Brewer CAP theorem [8], formalized by Gilbert and Lynch [20], states that "A distributed computer system can simultaneously provide only two of three desirable properties: *Consistency*, *Availability*, and *Partition tolerance*". This suggests that it is difficult to support ACID transactions in the cloud environment where distributed data and high availability are essential elements. As testimony to the influence of the CAP theorem, many cloud providers have pretty much abandoned transactional support for data spanning multiple nodes. Dynamo [17], BigTable [10], Facebook Cassandra [24], Windows Azure [29], and PNUTS [11] all stop short of providing general purpose transactions. Instead, these systems opt for *availability* by relaxing *consistency*, supporting either (a) atomicity over only a single data item or a collection of items, or (b) eventual consistency [34], i.e., data updates become visible to everyone after a finite time. Weak consistency is sometimes acceptable. Examples of applications that can tolerate weak consistency include keyword search, inventory search, and setting user preferences or recommendations (e.g., Facebook "Like" options, or movie ratings). On the other hand, many applications, new and old, would like to use cloud storage, yet find that weak consistency makes life very difficult. Examples of new applications include social networking and Web 2.0 applications, online auctions, and collaborative editing. Old applications include traditional database services such as credit card transactions and flight reservations.

To date, three primary approaches have explored providing ACID transactions in a cloud environment: (a) relying on application developers to implement their own consistency checks [7], which is both burdensome and inefficient, (b) making databases and data storage elastic to scale up and down with the current workload [12, 13, 14, 33], which still has either limited scalability or limited transaction support, and (c) extending single-key transactional support to multi-key [9, 15, 19], which is still limited in terms of not supporting transactions over keys in different groups.

This paper describes the architecture and functionality of the Deuteronomy system that provides efficient ACID transactions for data anywhere, including the cloud. Deuteronomy distinguishes itself from previous efforts by its radical approach of factoring the functions of a database storage engine kernel into: (a) a transactional component (TC) that provides transactions via "logical" concurrency control and undo/redo recovery but does not know physical data location and (b) a data component (DC) that caches data and knows about the physical organization, e.g. access methods, and supports a record-oriented interface with atomic operations, but knows nothing about transactions. Applications submit requests to the TC. The TC uses a lock manager and a log manager to logically enforce transactional concurrency control and recovery. The TC passes requests to the appropriate Data Component (DC). The DC, guaranteed by the TC to never receive conflicting concurrent operations, need only support atomic record operations, without concern for transaction properties that are already guaranteed by the TC.

A salient feature of Deuteronomy is the ability for transactions to span multiple DCs. For example, consider a transaction that updates two tables: an order table stored at a DC hosted on a local enterprise server, and a payment table stored at another DC hosted in the cloud. A client executes this transaction at a single TC, without specifying the physical location of the data. The TC performs all operations necessary for transactional support (e.g., logging/locking), and routes the data update operations to the correct DC ei-

§Work done while at Microsoft Research, Redmond, WA

5th Biennial Conference on Innovative Data Systems Research (CIDR) January 9-12, 2011, Asilomar, California, USA..

ther at the enterprise server (i.e., an order update) or in the cloud (i.e., a payment update). Upon completion, the TC is responsible for committing the transaction and ensuring the updates are stable at both DCs.

The deuteronomy architecture is scalable in three dimensions. (1) From a user perspective, if more application execution capability is needed then more applications servers (i.e., TC clients) can be added that interact with a single TC. (2) If data volume grows, more DC servers can be added "underneath" a TC to handle the storage and data manipulation workload. (3) For the case that transaction rates reach a degree that saturates the computational resources of a single TC servicing multiple clients and interacting with multiple DCs, multiple TCs can be instantiated (on separate machines) supporting transactions on disjoint sets of data. Thus, workloads can be split between TCs as long as the data they update is disjoint. However, we believe a single TC is capable of handling large workloads, as its performance-intensive operations consist mainly of locking, logging, and communication overhead. For OLTP workloads (for which Deuteronomy is intended) that are update intensive and do not deal with large answer sets, communication bandwidth should not be a system bottleneck until transaction rates saturate the communication link. Similarly, the execution load (i.e., logging and locking) should not be substantial at the TC node until transaction rates are enormous.

A Deuteronomy TC can be seen as providing *transactions as a service*. Storage systems in the cloud can "outsource" their transaction services to a TC. Alternatively, a TC can be seen as a storage engine that "outsources" its physical data storage management to another component (DC) in the cloud. Careful separation of transaction and data services enables multiple deployment scenarios. A TC can be applied to data *anywhere*, e.g., in one cloud, in multiple clouds, local, or at an enterprise server. Further, a TC can allow a transaction to spread over multiple DCs. Multiple TC deployment scenarios are also possible. Both TC and DC can be at a client and access local data, a TC can be at the client while the DC is in the cloud, or both the TC and DC can be cloud-based.

Previously [26, 27, 28], we described how to separate transactional functionality from data management functionality. This motivated our current effort, where we view cloud storage as an enormous atomic key-value store where data accessed within a transaction need not be co-located on an individual node to enable ACID transactions. The contributions of the current work are (1) the architecture of our multi-threaded TC; (2) a new TC:DC protocol in which the DC executes operations prior to TC logging them; (3) a new implementation of log control operations to deal with this protocol; and (4) initial experiments using an adapted TPC-W workload [32] that demonstrate both good performance and the impact of cloud latency on performance.

The rest of this paper is organized as follows. Section 2 highlights related work. The Deuteronomy system architecture is presented in Section 3. Section 4 provides details of the TC internals. Section 5 discusses transaction optimizations, while Section 6 provides an end-to-end example of a transaction executing in Deuteronomy. Section 7 discusses DC deployment scenarios. Preliminary performance results are provided in Section 8. Section 9 discusses availability. Finally, Section 10 concludes the paper, and Section 13 discusses a demonstration of the Deuteronomy system.

2. RELATED WORK

Several approaches have been proposed to enable ACID transactions for the cloud. This has been motivated by: (a) the lack of transactional support in existing commercial and open-source cloud services (e.g., [3, 9, 10, 11, 17, 19, 21, 24, 29]), (b) the emergence of new applications that require transactional support in the cloud, e.g., Web 2.0 applications, social networks, and collaborative editing [4, 23], and (c) the desire for traditional database applications, e.g., credit card transactions and flight reservation, to make use of the cloud infrastructure. The approaches can be divided into the following three broad categories:

Application-provided consistency. This approach relies on application developers to be responsible for ensuring transactional consistency. It puts a huge burden on application developers. It may also incur a big performance hit due to the need to call the server multiple times to ensure consistency [7]. While operations that do not require strong consistency guarantees can be realized efficiently, for others needing strong consistency, this performance impact may be unavoidable. This approach makes sense only for applications that have a very low fraction of transactions that require strong consistency guarantees [23].

Localized transaction support. Google Megastore [6] and Microsoft SQL Azure [9] support transactions over multiple records, however, they require that these records be co-located in some way. Developers must cluster Megastore data items into hierarchical groups. For SQL Azure, database size is constrained to fit on a single node. For larger data sets, an application needs to partition the data among different database instances. ElasTras [14] is an elastic database design that scales up and down with the transaction workload, but does not provide transactions upon recovery and supports a restricted transaction semantics, termed minitransactions [2], that execute within one data partition.

Limited wider transaction support. G-Store [15] allows for dynamic group formation (a relatively costly operation), where transactions are not allowed across these formed groups. CloudTPS [35, 36] assumes that transactions are short-lived and only access well-identified items (a group). Then it employs a two-level hierarchy of transaction managers running a global two-phase commit protocol over a set of transactions that each access a single item. The ecStore [33] is an elastic distributed storage system with three layers: storage nodes, replication layer, and a transaction layer that provides a hybrid design of multi-version and optimistic concurrency control. Transactions need full knowledge of data layout in the storage nodes.

Deuteronomy distinguishes itself from these approaches in architecting a complete separation of transaction services from data services. Such modular design allows for porting the Deuteronomy TC over any local or cloud data storage. The Deuteronomy TC can provide transactions across data anywhere, in the cloud or locally, not limited to a specific node, nor requiring special setup costs. This transaction support is transparent from both application developers and DCs, making their life much easier when dealing with the "elastic storage" provided by the cloud, though including DC caching and log synchronization functionality on top of some cloud infrastructures does take some effort.

3. OVERALL SYSTEM ARCHITECTURE

Figure 1 gives the Deuteronomy architecture depicting one transaction component (TC) (the large gray rectangle) and multiple data components (DC) with their physical storage as either local or in the cloud. The TC has five main components, depicted as white rectangles: *session manager*, *record manager*, *table manager*, *lock manager*, and *log manager*. The TC-DC interaction contract [27] is enforced by a set of control operations (depicted by dark rectangles in both the TC and DC) that mainly ensure recovery after failure of TC and/or DC, e.g. the write-ahead log protocol. The DC needs to manage its own data and storage, and can do so any way it likes as long as it supports atomic record operations and the "other side" of the control operations. This is a strong feature of Deuteronomy, as it makes the TC portable to many data storage providers and describes precisely what is needed on the DC side, i.e., the atomic record and table operations, the control operations, and the interaction contract.

Implementing a DC is non-trivial, as the DC provides both cache management and access method support, and in addition, it must fulfill the TC-DC contract, which includes control operation support, guaranteeing idempotence of operations, and recovery. A record-oriented cloud infrastructure is used by a cloud DC as if it were record-oriented disks. The important thing here from the transactional viewpoint is that there can be multiple DCs, the DCs can be located anywhere, the data managed by a DC can itself be "scattered across" the cloud or be local, and yet a single TC can effectively provide transactions for applications under any of these circumstances.

Applications submit their requests directly to the TC. These requests are handled by a multi-threaded *session manager*. The first request initiates the session, and the session manager establishes an authenticated session for the user. Subsequent requests flow through the session manager and are dispatched to the other components. Based on whether the request is for a record operation (e.g., read/write record) or table operation (e.g., create/delete table), the TC "session" invokes either its *record manager* or *table manager*, respectively. In both cases, the record/table manager calls both the lock and log manager to perform "logical" concurrency control and recovery.

We limit TC knowledge of threading to (1) the session manager, which does all thread management, (2) the lock manager which has to arbitrate and protect its lock data from race conditions and occasionally needs to block a thread when transactions have conflicting accesses, and (3) the log manager, which needs similar protection for its data structures, and occasionally needs to block a thread while log records are forced. Importantly, both table manager and record manager are thread safe without needing to be aware of threading issues (i.e. they are "thread oblivious").

Resources within the TC are all treated as logical data items in that their identification does not include physical location information. Locks are taken without knowledge of the physical layout of the stored data. Similarly, the log manager posts log records with resources described logically and without physical location information. These logical resources are mapped via metadata stored via the *table manager* to identify which DC owns the requested data. Metadata can be added throughout the lifetime of the TC in a similar way as done with traditional database catalogs.

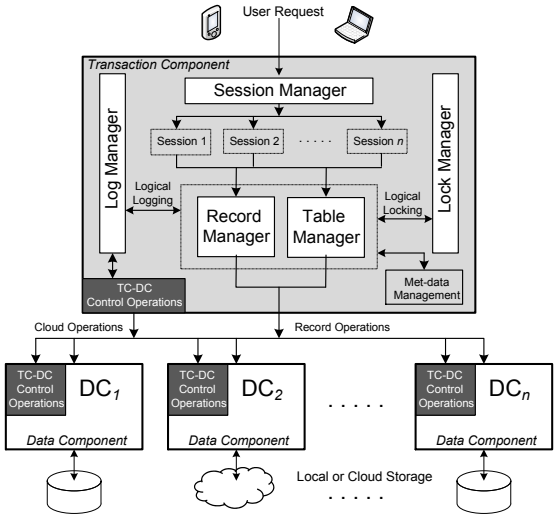


Figure 1: Deuteronomy System Architecture

Using its metadata, the TC sends table or record operations to the appropriate DC. The DC executes each of these operations as a stand-alone atomic operation, without worrying about any transactional conflicts among concurrent requests as TC locking will guarantee the absence of such conflicts. The TC also logs the operations as they are successfully completed (i.e., after the DC returns). This sequence of locking, forwarding an operation to a DC, and then logging, is quite different from our original thoughts [27], and will be elaborated in the record manager section.

There are three important points to emphasize:

1. Data can be stored *anywhere*, e.g., in a local disk, in a flash device, in the cloud, etc. TC functionality in no way depends on where the data is located.
2. The TC and DC can be deployed in a number of ways. Both can be located within the client, and that is helpful in providing fast transactional access to closely held data. The TC could be located with the client while the DC could be in the cloud, which is helpful in case a user would like to use its own subscription at a TC service or wants to perform transactions that involve data in multiple locations. Both TC and DC can be in the cloud, which is helpful if a cloud data storage provider would like to localize transaction services for some of its data to a TC component.
3. There can be multiple DCs serviced by one TC, where transactions spanning multiple DCs are naturally supported because a TC does not depend on where data items are stored. Also, there can be multiple TCs, yet, a transaction is serviced by one specific TC.

4. TRANSACTION COMPONENT (TC)

This section discusses the five major components of the TC: the *session manager*, *record manager*, *table manager*, *lock manager*, and *log manager*, the ones on the most performance sensitive execution path.

4.1 The Session Manager

The *session manager* is the application facing module of the TC, providing the interface to the application (henceforth referred to as user) and overseeing the execution of

its requests. The session manager maintains communication connections to users and provides multiplexed use of these connections. It authenticates a user when a session is initiated. Each session can support a stream of transactional requests, though within a session, there are no concurrent transactions. Hence it is sessions that are assigned to threads, with a session never using more than a single thread.

The session manager maintains a thread pool, a thread being dispatch for a session when a request from that session arrives and there is no thread assigned for this session. A dispatched session thread handles session requests in sequence. After serving a request, the session thread finds the next unprocessed queued request in this session and executes it. Queued requests are possible since a client can issue a batch of ordered requests. If there is no waiting request for the session, the thread is returned to the thread pool so that it can be used to handle requests from another session. Maintaining a thread pool permits fast request handling without the overhead of thread creation.

The session manager oversees the execution of user requests, calling the record manager and/or table manager as needed. User requests may be bracketed with explicit Begin/Commit transaction operations or use implicit transactions in which the session manager will provide these operations when a request from a client is not explicitly bracketed. In any case, a user (session) has only one open transaction at a time. User requests may result in multiple calls to table and record manager, e.g. a record update may require that the table manager be accessed to interrogate the catalog to identify the DC at which the data is managed; followed by an invocation of the record manager to perform the data manipulation operation. The session manager also marshals and de-marshals requests and replies between client and TC.

4.2 The Record Manager

The *record manager* supports operations that include modifying or reading records from DCs. Record operations "logically" coordinate with both the lock and log managers without knowledge of physical data placement and, in some cases, of the values of keys at the DC. In general, the *record manager* is responsible for two classes of operations: (a) reading operations that include reading a single record or a range of records, and (b) writing operations that include inserting, updating, and deleting a single record.

4.2.1 Read Operations

For read operations, the record manager first requests from the *lock manager* the appropriate lock(s) on the requested resource(s) (detailed in Section 4.4). Once the relevant locks are granted, the read request is forwarded to the DC either for a single record or for a range of records. Reads are not logged.

A user-provided key identifies a record for a singleton record operation. However, this is not the case for range reads, where the boundary keys that bracket the records of the range need not be real key values associated with existing records. Thus our "range" locking needs to be done without knowledge of the key values of records in the range. This is discussed in the lock manager subsection 4.4.

4.2.2 Write Operations

Write operations need to be both locked and logged. Both locking and logging are done with neither physical data placement information nor knowledge of surrounding keys. Both locking and logging requirements caused us to re-think the nature of the TC:DC interface, with the result that we changed the expected protocol specifics introduced in [27] to improve performance.

Locking: We do not exploit "next key" range locking, even of the type described in [28]. We made a strategic decision that even if we could afford to set such key value range locks, we could not afford to test them during inserts and deletes. These operations would need to know the "next key" value to test the "next key" lock protecting the gap (range) between keys for a range reader. To learn the "next key" requires that we read it from the DC, but this doubles the number of round trips to the DC needed for every insert and delete. While this may be reasonable when the DC is local, it is not when dealing with remote DCs with large latencies.

Logging: There is no problem with logging logical record identifiers for log records, and earlier work [26] demonstrated that doing this would produce at worst a modest reduction in speed of recovery, and little impact on normal execution. There are two problems with logging addressed in Deuteronomy that are of a different nature.

1. We had modeled our TC:DC interface [27] on the recovery guarantees framework developed for application recovery [5]. This posted information to the log prior to sending messages to ensure that the sender remembered a message it sent whenever a receiver remembered it (causality). But if we logged requests, we would need to also log replies in order to know when an operation succeeded. And not all operations (requests) succeed. For example, inserting a record when a record with the same key already exists or updating a record that does not exist are errors. We want to log operations only once. To do that, we need to know their outcome at the point when written to the log.
2. A second problem is that transactional logging requires both before and after state so that operations can be undone [31]. When an insert succeeds, the before state is null, so undo logging for inserts requires nothing new. For deletes and updates however, we did not want to be required to read the record before we changed it. By waiting for the DC to execute the operation and return to the TC prior to logging, we can have the response from the DC include the before image of the record which we then can include in our log records.

The preceding problems caused us to want a record manager protocol that orders locking and logging activities as follows:

1. Request appropriate locks, and generate a log sequence number (LSN), in monotonically increasing order, for a DC data modification request. The LSN is generated only after the appropriate write lock is granted. LSNs are not generated for read or intention locks. Our lock manager guarantees the order of these LSNs is consistent with the conflict order of the operations.
2. Send the operation to the DC for execution. The DC uses the LSN as it would in a conventional setting to identify the operation and provide recovery idempotence. For update and delete operations, the before

image is returned. For all operations, an indication of whether they succeeded is returned.

3. Log the operation after the DC has returned having executed the operation, again using the lock manager provided LSN to identify the operation. During recovery, the LSN for a successful operation is sent again to the DC along with the operation for the DC idempotence test.

This protocol is possible only because the TC:DC interface includes control operations which can be used to enforce causality and permit recovery management in this new setting. In particular, the TC can, via the control operations, specify when the DC can make information stable. It does this via an EOSL ("end of stable log") call. Operations with LSNs larger than the latest LSN provided to the DC via an EOSL call must be "forgettable".

Note here that when using this protocol, the LSNs on the log may be out-of-order as there is no guarantee about which requests will be granted/acknowledged first due to the multi-threaded nature of both TC and DC. However, what we are sure about is that conflict order will be preserved through all LSNs in the log file. This is guaranteed by the *lock manager*.

4.3 The Table Manager

The *table manager* is mainly concerned with data definition language (DDL) operations on a table that include creating and deleting tables, as well as creating, deleting, and modifying table columns. These operations sync with the lock and log managers and are passed to the appropriate DC in exactly the same way as the update operations described for the *record manager* in Section 4.2. In addition to executing DDL operations, the table manager has two other responsibilities: *meta-data management* and *creating and altering logical locking partitions*; we now describe these responsibilities in detail.

4.3.1 Metadata Management

The table manager is responsible for maintaining two primary metadata catalogs: (1) A *table catalog* stores an entry for each table containing its name, owner information, and the DC that stores the table. This catalog is primarily used to direct read/write requests to the appropriate DC. (2) A *column catalog* stores table column information including column name, constraints (e.g., primary/foreign key), and minimum and maximum values. Each TC stores its metadata catalogs at a *master* DC, a designated "default" DC whose address is given to each TC upon initialization and kept safe and persistent. When a TC subsequently restarts (e.g., due to a crash), it uses the stored master DC's address to retrieve its catalogs.

Like regular table and record operations, all operations that modify metadata catalogs synchronize with the lock and log managers. In fact, to ensure consistency between tables and their metadata, we wrap each table operation (e.g., create table) in a transaction with a symmetric operation that manipulates the appropriate metadata catalog (e.g., adding an entry to the table catalog).

4.3.2 Logical Partition Creation

As will be discussed in detail for the lock manager, there is a need for logical locking partitions in Deuteronomy. Essentially, the TC knows *only* about table and record lock resources, but not about the pages on which records or tables are stored. Pages have served as an intermediate lock resource in traditional database systems. But, in Deuteronomy, the TC cannot lock pages since they are not known to it. To compensate for this, the table manager defines *logical partitions* as an intermediate granularity lockable resource for each Deuteronomy table. A straightforward approach we currently use to create logical partitions is to divide the key range equally into a fixed number of partitions. However, more sophisticated techniques can be used to provide more uniform coverage by data volumes. New partitioning techniques can be employed without affecting the overall operations in Deuteronomy.

4.4 The Lock Manager

The *lock manager* is called from either the table or record managers to obtain the appropriate locks before user requests are forwarded to the DC. The lock manager must be thread aware as its lock tables can be accessed by many session threads concurrently. It uses a monitor to protect the lock table entries. It causes a session thread to block (sleep) if it encounters a conflicting lock. This part of the lock manager is quite traditional. Our lock manger also supports requests for multiple locks, with the lock manager returning after all lock requests are granted.

We employ multi-granularity locking with three levels of resources: table, partition, and record. Our multi-lock requests are used to request known locks down this hierarchy so that a single call is sufficient to acquire a record level lock. For table or record level locks, it is straightforward to provide table ID(name) and record ID(key), respectively, which are known to the TC as well as the DC. The TC knows these via the user request and its metadata.

Partitions are present in our multi-granularity hierarchy to facilitate reading ranges of records [28]. For example, consider the query that selects all employees with IDs from 10 to 40, and runs with serializable isolation. The lock manager needs to lock all the keys in the range [10,40]. However, since the TC knows nothing about the stored data, it has no way of locking these records without first reading them from the DC, which is very expensive in a cloud setting. Instead, the record manager will consult the table manager, which will return to it with a partition ID. The table manager is responsible for knowing about key domains and can partition them "logically" as appropriate.

The TC record manager will utilize these logical partitions to request locks that cover the requested range through locking all the logical partitions that overlap with the requested range. With logical partitions, reading/writing a single record requires an intent lock on the table and the partition resources that cover the requested record, and an explicit lock on the record itself. Reading a range of records requires an intent lock over the table, then, a set of explicit locks on all the logical partitions that overlap with the requested range. Individual records are never locked for a range read. Note that this is a different protocol than used in [28], where individual records were locked in "border" partitions. We felt it essential to avoid checking a next key lock during insert and delete operations, which is required for key range locks. Using only partition locks, which are analogous to page locks, is a cruder approximation to the range of records, but it avoids this extra "next key" access.

As discussed in Section 4.2, the lock manager is also re-

sponsible for generating LSNs to be used when logging writing operations. The main idea is that LSNs generated by the lock manager are guaranteed to be in conflict order. Thus, they are ideal for communicating with the DC and for providing idempotence for write operations at the DC. They do cause a complication at the log manager, however, which we describe next.

4.5 The Log Manager

The core of the log manager is conventional. Indeed, in our implementation, we used the Windows Common Log File (CLF) [30] as the TC transactional log. CLF natively supports multi-threading. Our code that wraps the CLF invocations also must deal with explicit threading, and appropriately synchronizes access to private data structures (e.g., thread-safe hash tables). However, we differ from conventional logging in that (1) we must deal with LSNs that are stored somewhat out-of-order on the log; and (2) we need to coordinate log management at the TC with cache management at the DC.

Recall that our protocol for dealing with a record operation first acquires a lock at the lock manager, with an LSN issued for it in strictly monotonic and hence conflict order. The request is then forwarded to the DC for execution, and we log the request only after the operation succeeds at the DC and returns control to the TC. Given the extensive use of multiple threads at all system levels, the LSNs, ordered when they leave the lock manager, can arrive out-of-order at the log manager. The log manager does not wait for LSNs of missing requests but rather writes the log record describing an operation immediately to the log.

Deuteronomy's recovery protocol has been described in some detail in an earlier paper [27]. Deuteronomy requires some changes to the more traditional ARIES [31] style algorithm. However, it shares the log management aspects with ARIES. That is, we need to enforce the write ahead log protocol, and we need to determine at what log position to begin our redo scan. These two aspects of recovery are normally done within a single database kernel, as part of an integrated algorithm. For Deuteronomy, however, managing the transaction recovery log is done at the TC, while cache management is done at each DC. Hence, log management requires that log and cache management be coordinated between TC and DC for successful recovery.

For the above reasons, the TC log manager will send two control operations to the DC: to enforce the write-ahead log protocol (causality); and to enable it to truncate the active part of the log via a checkpoint so that redo recovery time can be bounded and log space recovered and reused. Both operations are implemented as separate background threads, and do not interfere with the session-oriented record/table operations.

EOSL: The TC log manager periodically sends to each DC an LSN (denoted $eLSN$) indicating the *End Of Stable Log*. This operation permits a DC to write updates that it has cached back to stable storage. Before receiving an $eLSN \geq LSN$ of a cached update, the DC must not make that update stable. This permits it to "forget" the update ("forced amnesia") should the TC crash and lose the log tail containing that log record.

RSSP: At less frequent intervals, the TC log manager sends to each DC an LSN (denoted $rLSN$) indicating its desired *Redo Scan Start Point*. This operation requires that the DC write to stable storage all updates with LSNs earlier than $rLSN$ prior to returning from this operation. When control returns to the TC, the TC writes the $rLSN$ into its checkpoint information on its log, and uses the last written $rLSN$ as the start point for its redo scan should recovery be needed.

Given that LSNs are not ordered monotonically on our log, both $eLSN$ and $rLSN$ are "low water marks". That is, for EOSL, an $LSN \leq eLSN$ is stable. But an $LSN > eLSN$ may also be stable. For RSSP, the TC promises that every operation with an $LSN \geq rLSN$ will be replayed during recovery. Further, there is no guarantee that an operation with an $LSN < rLSN$ will be replayed during redo, so the DC must promise to make those operations stable before ACKing an RSSP call. This is the case even though some of these operations may, in fact, be re-sent to the DC because they appear later than the log position the TC uses for its redo scan start point.

4.5.1 EOSL

For EOSL, we need to ensure that we have received replies for all operations up to and including the operation with $LSN = eLSN$, and that log records for these operations are on the stable log. We may have received replies for some requests with an $LSNs > eLSN$, but we can ignore them. We keep a vector LSN-V (starting at the last value for $eLSN$) indexed by LSN. Each element contains the log position LP at which the operation with the given LSN is placed. Log positions are monotonic such that when a log record is posted to the log, it has a log position higher than all preceding log records. We maintain a current log position cLP to identify the log record slot in the log where next log record will be written.

When an operation identified with an LSN $oLSN$ returns from the DC and arrives at the log manager, its log record is placed in the current log position cLP. cLP is then stored at LSN-V[$oLSN$]. cLP is then incremented to reference the next position in the log buffer. We also track the log position of the end of the stable log, called sLP, which is updated whenever a flushed log buffer is stably written.

To determine a new $eLSN$, we scan LSN-V from the old $eLSN$ position until we reach an LSN-V[lsn] that is not set (operation has not yet returned and been logged) or LSN-V[lsn] $> sLP$. $lsn - 1$ becomes the new $eLSN$ as we now know that lsn is the lowest LSN not on the stable log.

4.5.2 RSSP

Implementing RSSP would seem to be easy since all we need is to direct the DC to make operations earlier than $rLSN$ stable. But the TC itself starts its redo scan at a log position rLP. For a pair of $rLSN$ and rLP, we require two things. (1) Any $LSN \leq rLSN$ must be stable on the log. This is the same condition as for $eLSN$, suggesting that we use an earlier $eLSN$ as our $rLSN$. (2) Since we are starting our redo scan at rLP, we need to make sure that we see for redo all operations with LSNs greater than $rLSN$. Thus we need to ensure that no operation with an $LSN > rLSN$ has a log position $LP \leq rLP$.

We extend our work for EOSL to help us with RSSP. We keep track of the maximum LSN that we have seen up to the time we scanned LSN-V for $eLSN$, called $maxLSN$,

and also remember the sLP (end of stable log) used in determining $eLSN$. We retain this information for the last several EOSL requests (since the prior RSSP) in a list of its own called EOSL-L. An element of EOSL-L consists of $\langle eLSN, sLP, maxLSN \rangle$, where the EOSL-L elements are ordered consistent with the time of an EOSL request, EOSL-L[i] being at an earlier time than EOSL-L$[i+1]$.

When an RSSP operation is invoked to determine a new $rLSN$, we set rLP = EOSL-L$[oldest].sLP$. We then scan EOSL-L$[i]$ entries beginning with $i = oldest + 1$. Any $LSN \leq$ EOSL-L$[i].eLSN$ is stable on the log. When EOSL-L$[i].eLSN \geq$ EOSL-L$[oldest].maxLSN$, we know that if the DC makes stable all operations with $oLSN \leq$ EOSL-L$[i].eLSN$ then all operations on the log before EOSL-L$[oldest].sLP$ will be made stable since none has an $LSN >$ EOSL-L$[i].eLSN$. Hence there will be no operations preceding $EOSL - L[oldest].sLP$ on the log that need redo when we set $rLSN$ = EOSL-L$[i].eLSN$.

Once we choose $rLSN$, we improve rLP by scanning EOSL-L entries EOSL-L[j] beginning with EOSL-L$[oldest]$. We stop at the largest j that satisfies $rLSN \geq$ EOSL-L$[j].maxLSN$. EOSL-L$[j].sLP$ is then used as the final rLP. The correctness of this improved rLP can be shown in the same way as the previous one.

5. TRANSACTIONAL OPTIMIZATIONS

Deuteronomy incorporates classical and important transactional optimization techniques. In this section we discuss how fast commit and group commit [18] optimizations are provided in Deuteronomy to improve throughput.

5.1 Fast Commit

Fast commit optimization allows a transaction to release all its locks before waiting for its commit record to be flushed to stable storage. Deuteronomy adopts the fast commit optimization by arranging the key steps that a transaction takes during commit as follows.

1. Create the commit record and post it to log buffer.
2. Release transaction locks.
3. Wait until transaction commit log record is flushed.
4. Remove the transaction from the table of active transactions.
5. Send reply to the client indicating the transaction has committed.

This order does not affect the correctness of the system since a transaction that releases its locks is guaranteed to commit earlier than any other transaction waiting for the locks. The guarantee comes from the fact that the earlier transaction places its commit record in log buffer before it releases its locks, and the log records are written sequentially during flush. So if a later transaction using the unlocked resources commits, it will be done only if the earlier transaction also commits. Compared to the original scenario where locks are released after the commit record is flushed, fast commit reduces lock wait time for every transaction.

To make this work correctly, we need to synchronize read-only transactions with this optimization. This can be done by writing commit log records for all transactions, including read-only transactions.

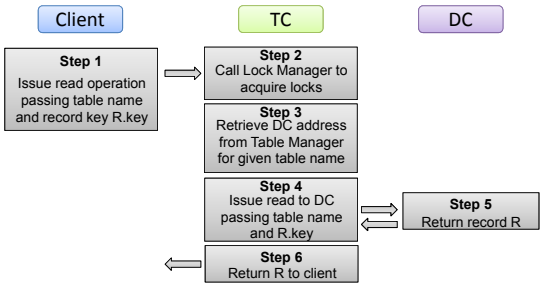


Figure 2: End-to-end steps for single record read

5.2 Group Commit

The group commit optimization delays a set of committing transactions for a small time period and commits them as a group by flushing the log buffer containing their commit records to the disk. For durability, a transaction is not committed until its commit log record has been flushed to the stable log. Without group commit, each commit triggers a system call to CLF to flush the log buffer to disk immediately. This requires CLF to flush the log buffer (write to the disk) for every transaction commit. Thus, every transaction incurs disk write latency, and log buffer storage utilization suffers.

Group commit amortizes the cost of a log force by grouping transactions that commit close in time and issuing a single log flush request for all transactions in the group. In Deuteronomy, this is achieved by having the thread of a committing transaction wait inside the log manager for a log flush. (This occurs at step 3 above.) A log flush writes all buffered log records to stable storage. This log buffer flush occurs either (1) when the buffer is full, or (2) after a small time delta, which is configurable, that enables the buffer to fill. In this way, we reduce the number of log I/O system calls from the number of commits to the number of commit groups by slightly holding back each committing transaction. All transactions of a group share the group commit write latency instead of each incurring the write latency.

6. AN END-TO-END EXAMPLE

To help demonstrate the technical details described so far, this section provides an end-to-end example of the steps necessary to execute a transaction in Deuteronomy. Our running example is the following transaction that reads and updates a single record R:

```
Begin Transaction
  Read record R
  Update record R with new value V
Commit Transaction
```

We describe the details of the four operations for this transaction in chronological order.

Begin transaction. To begin a transaction, the session manager first assigns a thread to the transaction (details in Section 4.1) and generates a unique transaction id, which is then used to initiate an entry in the transaction table that stores the log position of each active transaction's last operation (used to back-chain a transaction's log records).

Read record. The steps of a read operation for a single record are depicted in Figure 2. The client passes the TC

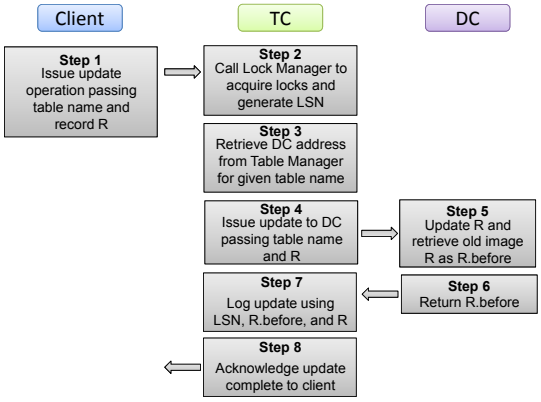


Figure 3: End-to-end steps for record update

the table name and record key R.key. Upon receiving the request, the TC calls the lock manager to acquire appropriate locks. The lock manager does not generate an LSN for these operations since reads are not logged in Deuteronomy. Next, using the given table name, the DC address is retrieved from the table manager, and the read request is sent to the DC passing the table name and record key. The DC then returns to the TC either record R if the read is successful, or an error if record R does not exist. The TC then passes back record R (or an error) to the client to complete the read operation.

Update record. The steps of an update operation are depicted in Figure 3. The client passes the TC a table name and updated record R. The TC then calls the lock manager to both acquire appropriate locks and retrieve an LSN for the operation. The DC address is then retrieved from the table manager, and the update is sent to the DC using the table name and record R. The DC first retrieves the before image of record R (abbr. R.before), and then performs the record update. The DC then returns R.before to the TC, and the TC logs the operation using the LSN, R.before as the before image, and R as the after image. Finally, the TC sends an acknowledgement of the update to the client, which completes the update operation. In the case of an error (e.g., record R does not exist at the DC), the TC will return an error to the client without logging the update.

End transaction. To end the transaction, the TC first writes a commit record to the log. Details of the Deuteronomy commit procedures are given in Section 5. Once the commit is logged, the transaction's entry is removed from the transaction table. Finally, the TC returns to the client and returns the transaction thread to the open thread pool.

7. DATA COMPONENTS (DCS)

We implemented and/or used a number of DCs: (1) a "stub DC" that merely returns immediately, used solely to test our TC code, (2) a "local DC" which keeps all data in main memory, also used in testing, (3) a "flash DC" that used a storage manager provided by the Communication and Collaboration Systems Group in Microsoft Research[1] that exploits flash memory, but is also usable with a disk for stability, and (4) a "cloud DC" written by a group in the Microsoft XTREME Computing Group[2] that uses Windows Azure storage to make data stable.

Our TC code is defined to permit us to use several DCs simultaneously, and the thread executing each TC operation simply invokes a DC operation as if it were local. Given our multi-threaded TC, this means that each DC needs to deal with multi-threading issues. When a DC is not local, there is a proxy DC behind the interface that forwards messages to the remote DC. In this case, it is the proxy that deals with the local (to the node of the TC) threading issues, such as commanding a requesting thread to sleep and waking it up when the reply comes back from the remote DC. A proxy DC is required for the cloud DC, which cannot be local.

Had we only intended using a TC with local DCs, we wouldn't have been so concerned about the need to read a record prior to locking records in a range or testing a next key lock when doing inserts and deletes. However, the cloud introduces large latencies (larger than a local disk). So it has been essential to tailor our TC:DC interface to minimize the number of times we incur cloud latency.

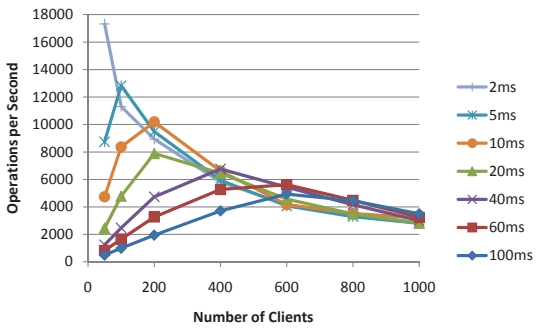


Figure 4: TC Performance for Varying DC Latencies

8. SOME PERFORMANCE RESULTS

8.1 Benchmarking

We evaluated performance via adapting a limited version of the TPC-W benchmarks [32] to work in the cloud environment. This is similar to the approach taken in [23, 22, 25]. Since we are most concerned about measuring TC performance, we show throughput under controlled changes in DC latency. The longer the latency to the DC, the higher the level of multi-threading we must employ to keep the processor busy. The more threads active at a time, the more collisions they will see, and perhaps lower cache hits as well. This results in higher throughput for lower latency deployments, as shown in Figure 4.

8.2 Improving Performance

We consider the performance reported for our benchmarking to be only respectable. We believe it is possible for performance to be substantially higher than we report above. Here we want to discuss the nature of the overhead introduced by the TC on the path to the data, and to understand what might be done to improve performance.

The first thing we did was to isolate TC performance from transactional aspects, in particular how much overhead was

[1]This group consisted of Sudipta Sengupta, Biplob Debnath, and Jin Li

[2]This group consisted of Roger Barga, Nelson Araujo, Brihadish Koushik, and Shailesh Nikam.

added by locking and logging. We ran tests with these capabilities disabled. Performance was only 20% better when locking and logging were disabled for the cloud latency case. As performance elsewhere improves, the cost of logging and locking will loom larger. But it is clear that performance gains will need to come from elsewhere.

We believe that there are two main impediments to higher performance. And these impediments are neither architectural nor are they intrinsic to the logic of operations as we have implemented operations. Rather, we believe there are two limitations that lie in the infrastructure that we used to build our prototype.

Threading: Operating system threads are much lighter weight than processes. However, whenever a system thread blocks or otherwise does a context switch, an OS thread crosses a protection boundary. This step adds substantial overhead to the cost of a thread switch. Because of this, SQL Server implements (in their SQL OS layer) user level threading called fibers. We did not use SQL OS in our implementation in the interest of rapid system prototyping. However, we believe that there is a substantial gain to be made by using fibers.

Implementation Language: We used C# as our implementation language because it reduced the programming effort. That was, we believe, a wise choice given the limited time we had to construct the system. But, for a "real" Deuteronomy deployment, we would have made a different choice. Building a system has a rough equivalence to working on the code in the inner loop of a large program. That is, system programming is much more performance sensitive than is application programming. Programming languages that are great for rapid prototyping and fine for application programming may be problematical for the inner core of system level programming. Deuteronomy (encompassing both TC and DCs) is part of that inner core. Indeed TC and DC together constitute the kernel of a database system. Thus, a language like C is the more appropriate language for a system to be widely deployed.

While we cannot quantify the performance that would result from using fibers and implementing in C, we expect substantial gains. At that point, we would need to revisit lock and log manager implementations. In the lock manager, we need to use more fine grained concurrency control for accessing the lock manager data structures, e.g. spin locks protecting smaller data extents. For the log manager, as with threading, performance would benefit from its code being entirely within user space to save the system protection boundary overhead of using CLF.

9. AVAILABILITY

Availability can be lost due to several forms of failure. Availability is maximized when the system can (1) minimize the extent of the availability loss when a failure occurs; and (2) reduce the time to recover from the failure causing the lost availability. In this section we describe a number of types of lost availability resulting from failures, and how the Deuteronomy architecture enables us to minimize the loss.

9.1 Data Unavailability

We focus first on data availability when the DC responsible for executing on the data has not failed. DC failures are considered below. A prime purpose of cloud infrastructures is to provide high availability. Data is typically replicated, with a consensus protocol used to ensure that a replica failure does not make data unavailable for update. So data unavailability should be a very rare event.

While there is nothing that Deuteronomy can do directly to make unavailable data available, accessible data need not become unavailable simply because it happened to be accessed in the same transaction as currently unavailable data. Such data can become transactionally consistent either by transaction commit or transaction abort. Both commit and abort are feasible depending on the specific state of each transaction. For transactions that are finished, commit releases locks on available data, while for abort, undo operations are first sent to the DC managing the data, and then locks are released. Neither outcome is a blocking outcome, and access to available data continues uninterrupted.

Further, a DC may be able to mask some data unavailability. This may enable some transactions to commit that would otherwise have aborted. So long as the data being accessed by transactions from the TC is in DC cache, availability continues. When the data becomes available again, the changes captured by the DC are written back to storage. Only when the DC needs to access data that is unavailable and not in its cache does it need to notify the TC. When this occurs, the TC aborts the transaction involved.

9.2 TC Failure

Availability in the presence of a TC failure depends on the robustness and accessibility of the TC transactional log. If the TC log is on a disk that is local to the TC, and is not itself replicated elsewhere, then the data handled via the TC is unavailable while the TC is down. However, if the log is available, e.g. cloud replication is used for the log, then the TC can failover to a standby TC that accesses the log. The standby TC initiates recovery using the log, and when recovery is complete, normal service resumes.

For a TC failure without an accompanying DC failure, recovery is very fast. Even with the TC log replayed from the RSSP, the DC has little recovery to perform as it has not crashed. The most substantive activity it needs to do is to reset its cache, removing updates that were not stable on the TC log at the time of the crash. Only data items modified by these updates, or ones reset along with them (e.g. if the cache was paginated and a reset affected other records on the page) need to be recovered.

9.3 DC Failure

Should a DC fail, a more expensive form of recovery is needed to bring the database back to the point where the failed DC can resume normal execution. We have outlined that recovery earlier [27]. The TC waits for the DC to come back up, and then initiates recovery with it. Because the DC cache has been lost, recovery now entails re-execution of all lost operations (i.e. redo recovery), and importantly, the DC cache needs to be re-populated with the active data as of the time of the crash to do this. While Deuteronomy recovery needs to be "logical", as shown in [26], recovery performance can be comparable to ARIES style recovery.

While it is conceptually possible to maintain a hot DC standby in the same way that a full-blown database systems maintains a standby, there is a negative to this. A database standby typically manages an independent database replica. Hence it can read and write to its replica and manage its cache independently of the primary database. To pursue

that strategy, we would need a hot standby DC to manage its own data replica. This can only be done by (1) replacing the normal cloud replication with our own replication so that a DC accesses a single replica or (2) having two replicated cloud data sets, each with a separate DC, thus adding another layer of replication on top of the normal cloud data replication. How best to maintain a hot standby using a single cloud replicated data set, where both primary and hot standby manage this data set is a topic for further work.

10. CONCLUSION

We have described our implementation of a TC that can provide transactional functionality over any storage infrastructure. To work effectively, we do need to provide a storage infrastructure with DC functionality in order to enable it to cache data and to post its results to stable storage lazily. The DC functionality also permits the TC to lazily force its log, and to truncate its log using normal database checkpointing methods.

Our TC provides transaction functionality, regardless of how the data may be distributed across the cloud. While accessing data in the cloud currently entails large latency, our TC nonetheless achieves decent performance. Most previous efforts have either tried to avoid cloud transactions, or have severely circumscribed their scope. Our TC implementation shows that this is not required, and that it is feasible to provide full ACID transactions and enable the applications that require them to be supported in the cloud.

11. ACKNOWLEDGMENTS

We want to thank our colleagues whose work on data components (DCs) enable this project to succeed. Our cloud DC was implemented by a group led by Roger Barga consisting of Brihadish Koushik, Nelson Araujo, and Shailesh Nikam. Jin Li and Sudipta Sengupta provided us with their FlashStore [16], which formed the basis of our Flash DC. An anonymous referee provided helpful remarks on our locking protocol.

12. REFERENCES

[1] D. Agrawal, S. Das, and A. E. Abbadi. Big Data and Cloud Computing: New Wine or just New Bottles? In *VLDB*, 2010.

[2] M. K. Aguilera et al. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP*, 2007.

[3] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[4] S. Amer-Yahia, V. Markl, A. Y. Halevy, A. Doan, G. Alonso, D. Kossmann, and G. Weikum. Databases and Web 2.0 panel at VLDB 2007. *SIGMOD Record*, 37(1):49–52, 2008.

[5] R. S. Barga, D. B. Lomet, G. Shegalov, and G. Weikum. Recovery Guarantees for Internet Applications. *ACM Transactions on Internet Technology, TOIT*, 4(3):289–328, 2004.

[6] Philip A. Bernstein. A blog entry about Google Megastore, published online on James Hamilton's Blog "Perspectives". http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx.

[7] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *SIGMOD*, 2008.

[8] E. A. Brewer. Towards Robust Distributed Systems (Keynote Talk). In *PODC*, 2000.

[9] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In *SIGMOD*, 2010.

[10] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[11] B. F. Cooper et al. PNUTS: Yahoo!'s hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, Aug. 2008.

[12] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational Cloud: The Case for a Database Service. Technical Report MIT-CSAIL-TR-2010-014, Massachusetts Institute of Technology, MIT, Mar. 2010.

[13] S. Das, S. Agarwal, D. Agrawal, and A. E. Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-4, University of California, Santa Barbara, UCSB, Mar. 2010.

[14] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud Workshop*, June 2009.

[15] S. Das, D. Agrawal, and A. E. Abbadi. G-Store: A Scalable Data Store for Transactional Multi-Key Access in the Cloud. In *SoCC*, 2010.

[16] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. In *PVLDB*, 2010

[17] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *OSDI*, 2007.

[18] D. J. DeWitt et al. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, 1984.

[19] J. J. Furman et al. Megastore: A Scalable Data System for User Facing Applications. In *Invited Presentation in SIGMOD Products Day*, June 2008.

[20] S. Gilbert and N. A. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.

[21] Hbase. http://hbase.apache.org/.

[22] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *SIGMOD*, 2010.

[23] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[24] A. Lakshman and P. Malik. Cassandra: structured storage system on a P2P network. In *PODC*, 2009.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *IMC*, Nov. 2010.

[26] D. Lomet and K. Tzoumas. Implementing Performance Competitive Logical Recovery. In *Under Submission*, 2010.

[27] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling Transaction Services in the Cloud. In *CIDR*, Jan. 2009.

[28] D. B. Lomet and M. F. Mokbel. Locking Key Ranges with Unbundled Transaction Services. *PVLDB*, 2(1):265–276, 2009.

[29] Microsoft Windows Azure. http://www.microsoft.com/windowsazure/windowsazure/.

[30] Microsoft Common Log File System: http://tinyurl.com/2fwlmux.

[31] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS*, 17(1):94–162, 1992.

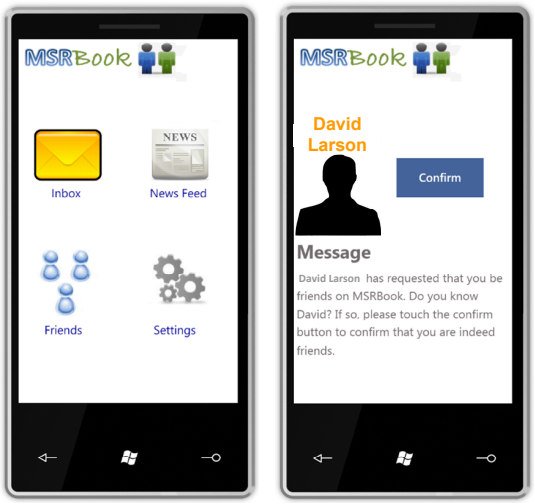
[32] TPC-W Benchmarks. http://www.tpc.org/tpcw/.

[33] H. T. Vo, C. Chen, and B. C. Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. *PVLDB*, Sept. 2010.

[34] W. Vogels. Eventually Consistent. *Communications of ACM, CACM*, 52(1):40–44, 2009.

[35] Z. Wei, G. Pierre, and C.-H. Chi. Scalable Transactions for Web Applications in the Cloud. In *Proceedings of the Euro-Par Conference on Parallel Processing*, 2009.

[36] Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. Technical Report IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010.



(a) Home screen (b) Friend request screen

Figure 5: Demo app: MSRBook social network

13. DEMONSTRATION DESCRIPTION

This section provides a demonstration description of the Deuteronomy system. We cover the demo applications, data, and configuration necessary to support unrestricted transactions for a social networking application that stores its data anywhere in the cloud.

13.1 Application

The application we use to demonstrate Deuteronomy is *MSRBook*, a cloud-based social-networking application. *MSRBook* comes in two versions: (1) *Mobile-based*, implemented as a Microsoft Windows Phone 7 application and depicted in Figure 5, and (2) *Web-based*, built for a standard web browser (screen-shot omitted due to space). Users perform actions in *MSRBook* similar to other well-known social networking applications, such as updating friend lists, posting items on friend feeds, and sending and receiving messages.

MSRBook uses Windows Azure cloud-based storage for managing its data. Deuteronomy provides transaction management for this data. *MSRBook* interfaces with the TC session manager and all transactions performed by the application use the TC interface methods covered in Section 4.2.

13.2 Data and System Configuration

For each user account, *MSRBook* stores two major pieces of data: (1) a friend list that stores who a particular user is connected with in the application, and (2) news feed items, containing news updates a user wishes to share with friends. *MSRBook* partitions its account data by user last name into three primary partitions: [a-f], [g-p], [q-z]. Each partition is managed by a separate DC, and is hosted on a different Windows Azure partition, meaning data for any two user accounts are *not* guaranteed to be co-located on the same Azure storage node. *MSRBook* interfaces with a single TC that provides transaction support for all three DCs as depicted in Figure 6. The TC, as well as all DCs, are cloud-based. The TC is implemented and hosted as a Windows Azure web role, while Each DC is implemented as a Windows Azure worker role (see [29] for details of web and worker roles).

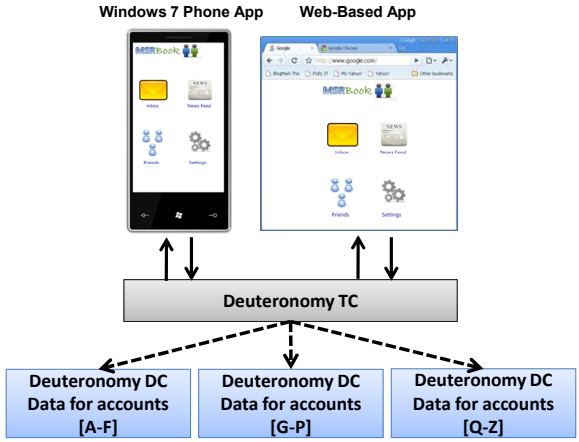


Figure 6: Demonstration scenario overview

This configuration is significant for two reasons. First, it mirrors a simple but realistic partitioning scheme typical for many cloud-based applications. Second, such a configuration does not guarantee that any two (or more) users are co-located on the same storage partition, meaning many existing cloud-based transaction approaches (e.g., Azure entity group transactions [29]) cannot provide support for a ACID transactions involving *any* two (or more) user accounts.

13.3 Demonstration Scenarios

Transactions in Deuteronomy. Our first demonstration focuses on the transactional details of users updating their friend lists in *MSRBook*. In this scenario, a user Larson notifies user Smith that he would like to connect as friends on *MSRBook*. Upon navigating to the friend notification screen (depicted in Figure 5 (b)), user Smith touches the "confirm" button in order to confirm his friendship with Larson. This action requires a transaction that updates both friend lists as well as both news feeds for the two users. Such a transaction requires only six lines of straightforward code in Deuteronomy as follows.

```
Begin Transaction
  Insert user Larson into friend list of Smith
  Insert new friend update into Larson's news feed
  Insert user Smith into friend list of Larson
  Insert new friend update into Smith's news feed
End Transaction
```

Given the configuration of the data (Larson and Smith exist in separate Azure storage partitions), such a simple and straightforward transaction is only possible in Deuteronomy. Under these partition constraints, all other cloud-based transactional support (see Section 2) require some form of eventual consistency that is orders of magnitude more complicated for application developers to implement [1].

Scalability. Our second demonstration provides a live showcase of scalability performance of Deuteronomy using the MSRBook application. Using a workload generator that simulates tens of thousands of simultaneous user friend update requests in MSRBook, we report live throughput numbers for Deuteronomy under such a workload. We also concurrently perform friend updates using the Windows Phone 7 application (Figure 5) while the simulated workload runs to show that response time is on the order of milliseconds.

Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective

Hiroshi Wada[*†], Alan Fekete[‡], Liang Zhao[†*], Kevin Lee[*] and Anna Liu[*†]
[*] National ICT Australia – NICTA
[†] School of Computer Science and Engineering, University of New South Wales
[‡] School of Information Technologies, University of Sydney
[*] {firstname.lastname}@nicta.com.au
[‡] {firstname.lastname}@sydney.edu.au

ABSTRACT

A new class of data storage systems, called *NoSQL* (Not Only SQL), have emerged to complement traditional database systems, with rejection of general ACID transactions as one common feature. Different platforms, and indeed different primitives within one NoSQL platform, can offer various consistency properties, from Eventual Consistency to single-entity ACID. For the platform provider, weaker consistency should allow better availability, lower latency, and other benefits. This paper investigates what consumers observe of the consistency and performance properties of various offerings. We find that many platforms seem in practice to offer more consistency than they promise; we also find cases where the platform offers consumers a choice between stronger and weaker consistency, but there is no observed benefit from accepting weaker consistency properties.

1. INTRODUCTION

Cloud computing is attracting interest through the potential for low cost, unlimited scalability, and elasticity of cost with load [7, 8, 11, 33]. A wide variety of offerings are typically categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS is exemplified by Amazon Web Services (AWS), and provides the capability to execute existing programs on a virtual machine that is essentially the same as a standard box with a standard operating system. The consumer has control over the virtual resources. Each PaaS system offers a distinctive set of functionalities as an API, that allow programs to be written specially to execute in the cloud; Google AppEngine (GAE) is an example of this approach.

In PaaS systems, a persistent and scalable storage platform is a crucial facility. In an IaaS environment, one could simply install an existing database engine such as MySQL in one's virtual machine instance, but the limitations (performance, scale, and fault-tolerance) of this approach are well-known, and the traditional database systems can become a bottleneck in a cloud platform [2, 27]; thus novel storage platforms are commonly offered within IaaS clouds too. These storage platforms operate within the cloud platform, and take advantage of the scale-out from huge numbers of cheap machines; they also internally have mechanisms to tolerate the faults that are inevitable with so many unreliable machines. Examples include Amazon SimpleDB[1], Microsoft Azure Table Storage[2], Google App Engine datastore[3], and Cassandra[4]. A term often applied to these storage platforms is *NoSQL* (Not Only SQL). NoSQL database systems are designed to achieve high throughput and high availability by giving up some functionalities that traditional database systems offer such as joins and ACID transactions. NoSQL data stores may offer weaker consistency properties, for example eventual consistency [32]. A client of such a store may observe values that are stale, not from the most recent write. This design feature is explained by the CAP theorem, which states that a partition-tolerant distributed system can guarantee only one of the following two properties: data consistency, or availability [17]. Many of NoSQL database systems aim for availability and partition tolerance as their primary focus and thus they relax the data consistency constraints.

It is a new challenge for developers to write applications that use storage offering weak consistency. For example, recent work by Hellerstein [19] has identified a class of monotonic programs that give correct results on eventual consistent data. The application designer therefore tries to express their computational task using only monotonic operations. Further complicating the programmer's task, there are variant consistency properties that may or may not be provided, such as read-your-writes, monotonic reads, or session consistency, each changing the set of possible situations, and thus what the code must be written to handle. The effort of coding for a weak consistency model is typically justified by pointing to corresponding tradeoffs, such as better availability, lower latency, etc [16].

We have experimentally investigated these issues, from the view of the consumer of the storage facilities. That is, we try to see what kinds of inconsistency are seen *in the results returned from operations*, and how frequently these situations arise. This contrasts with research on cloud-based

CIDR'11 Asilomar, California, January 2011

[1] aws.amazon.com/simpledb/
[2] www.microsoft.com/windowsazure/
[3] code.google.com/appengine/
[4] cassandra.apache.org/

storage platforms [9, 10, 12] that takes the view of the platform owner and focuses on algorithms and the properties of the data held in various replicas within the platform.

Our main contributions are detailed measurements over several storage platforms, that show how frequently, and in what circumstances, different inconsistency situations are observed, and what impact the consumer sees on performance properties from choosing to operate with weak consistency mechanisms. The overall methodology of our experiments, for measuring consistency as seen by a consumer, is another contribution. In Section 2 we report on the experiments that investigate how often a read sees a stale value. For several platforms, data is always, or nearly always, up-to-date. For one platform (SimpleDB), we often see stale data, and so in Section 3 we investigate more deeply the consistency properties of this platform, covering issues such as consistency among multiple data elements, and cases where operations on one element impact on reads of another element. Section 4 then explores the performance of different consistency options; in particular, we investigate whether the consumer is offered any tradeoff in cost or performance, to compensate for using weak consistency operations. Section 5 discusses some limitations to generalising our results. In Section 6 we connect and contrast our work with other research related to this topic. Section 7 gives some conclusions and suggests directions for further study.

2. STALENESS OF DATA

We first investigate the probability of a consumer observing stale data in an item.

Figure 1 illustrates the architecture of the benchmark applications in this study. There are three cloud-deployed roles: the data store, and two computations, *writer* and *reader*. A writer repeatedly writes 14bytes of string data into a particular data element; the value written is the current time, so that we can easily check which write is observed in a read. In most of the experiments we report, writing happens once every three seconds. A reader role repeatedly reads the contents from the data element and also notes the time at which the read occurs; in most experiments reading happens 50 times every second. In some of our experiments, we use one thread for the writer role and one or multiple threads each implementing the reader role, in other experiments we have a single thread that takes both roles. We refer to one "measurement" of the experiment as running the writing and reading for 5 minutes, doing 100 writes and 15,000 reads. We repeated the measurement once every hour, for at least one week, in October and November 2010. In a post-processing data analysis phase, each read is determined to be either fresh (if the value observed has a timestamp from the closest preceding write operation, based on the times of occurrence) or stale; also each read is placed in a bucket based on how much clock-time has elapsed since the most recent write operation. By examining all the reads within a bucket, from a single measurement run, or indeed aggregating over many runs, we calculate the probability that a read which occurs a given time after the write, will observe the freshest value. Repeating the experiment through a week ensures that we will notice any daily or weekly variation in behavior.

2.1 Amazon SimpleDB

SimpleDB is a distributed key-value store offered by Amazon. Each key has associated a collection of attributes, each with a value. For these experiments, we take a data element to be a particular attribute kept for a particular key (a key identifies what SimpleDB calls an item). SimpleDB supports (among other calls) a write operation (`PutAttributes`) and two types of read operations, distinguished by a parameter in the call to `GetAttributes`: *eventual consistent read* and *consistent read*. The consistent read is supposed to ensure that the value returned always comes from the most recently completed write operation, while an eventually consistent read does not give this guarantee. Our study investigates how these differences appear to the consumer of data.

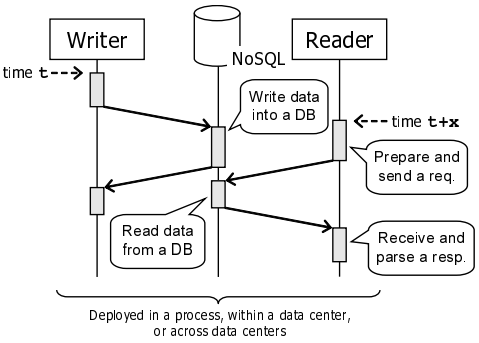


Figure 1: The Architecture of Benchmark Apps

SimpleDB is currently operated in four geographic regions independently (i.e., US West, US East, Ireland and Singapore) and each of them offers a distinct URL as its access point. For example, `https://sdb.us-west-1.amazonaws.com` is the URL of SimpleDB operated in US West. We used this as the data store for our experiments. Writer and reader are implemented in Java and run in EC2; they access SimpleDB in US West through its REST interface.

2.1.1 Accessing from a Single Thread

In the first study, we run the writer and reader in the same single thread on an `m1.small` instance provided by EC2 with Ubuntu 9.10. The writer/reader process is deployed in US West. We cannot be sure that the SimpleDB data store will be in the same physical data center as the computation [5], but using the same geographic region is the consumer's best mechanism to reduce network latency.

We executed a measurement run once every hour for 11 days from Oct 21, 2010. In total 26,500 writes and 3,975,000 reads were performed. Since we use only one thread in this study, the average throughput of reading and writing are 39.52 per second and 0.26 per second, respectively. (Each measurement runs at least five minutes.) The same set of measurements was performed with eventual consistent read and with consistent read.

2.1.1.1 Read-Your-Write Consistency.

Figure 2 and Table 1 show the probability of reading the fresh value plotted against the time interval that elapsed from when the write begins, to the time when the read is submitted. Each data point in the graph is an aggregate over all the measurements for a particular bucket containing all time intervals that agree to millisecond granularity; in the Table we aggregate further, placing all buckets whose time is in a broad interval together, and here we also show actual numbers as well as percentages. With eventual consistent read the probability stays about 33% from 0ms to 450ms. It

jumps up sharply between 450ms and 500ms, and it reaches 98% at 507ms. A spike and a valley in the first 10ms are perhaps random fluctuation due to a small number of data points. With consistent read, the probability is 100% from about 0ms onwards.

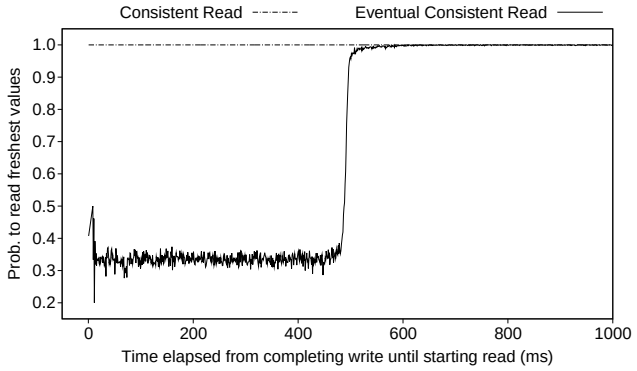


Figure 2: Probability of Reading Freshest Value

Table 1: Probability of Reading Freshest Value

Time Elapsed from Starting Write Until Starting Read	Eventual Consistent Read	Consistent Read
[0, 450)	33.40% (168,908/505,821)	100.00% (482,717/482,717)
[500, 1000)	99.78% (1,192/541,062)	100.00% (509,426/509,426)

A relevant consistency property is "read-your-writes", which says that when the most recent write is from the same thread as the reader, then the value seen should be fresh. As we find that stale eventual consistent reads are possible with SimpleDB within a single thread, so we conclude that eventual consistent reads do not satisfy read-your-writes; however, consistent reads of course do have this property.

We now consider the variability of the time when freshness is possible or highly likely, among different measurement runs. For eventual consistent reads, Figure 3 shows the first time when a bucket has freshness probability that is over 99%, and the last time when the probability is less than 100%. Each data point is obtained from a five minutes measurement run, so there are 258 data points in each time-series. The median of the time to exceed 99% is 516.17ms and coefficient of variance is 0.0258. There does not seem to be any regular daily or weekly variation, rather the outliers seem randomly placed. Out of the 258 measurement runs, 2 runs (0.78%) and 21 runs (8.14%) show a non-zero probability of stale read after 4000ms and 1000ms, respectively. Those outliers are considered to be generated by network jitter and similar effects.

2.1.1.2 *Monotonic Read Consistency.*

One consistency property that has been considered important [32] is "monotonic read", where a following operation sees data that is at least as fresh as what was seen before. This property can be examined across multiple data elements, or for a single element as we consider here. We find that consistent reads are monotonic as they should be, since each read should always see the most recent value. However, eventual consistent reads are not monotonic, and indeed the

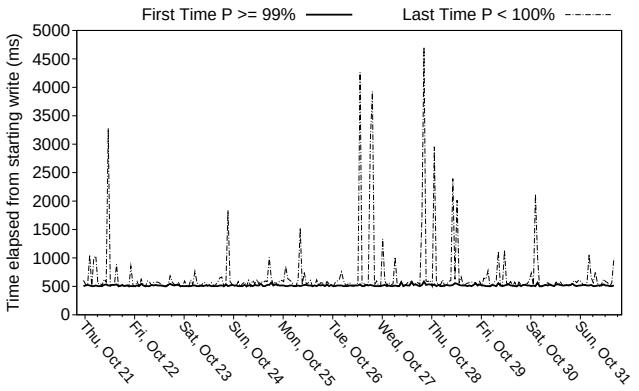


Figure 3: Time to See Freshness (Eventual Consistent Read)

freshness of a successive operation seems essentially independent of what was seen before. Thus eventual consistent reads also do not have stronger properties like causal consistency.

Table 2 shows the probability of observing fresh or stale values in each pair of successive eventual consistent reads performed during the range from 0ms to 450ms after the time of a write. The table also shows the actual number of observations out of 475,575 of two subsequent reads performed in this measurement study. The monotonic read condition is violated (that is, the first read returns a fresh value but the second read returns a stale value) in 23.36% of pairs. This is reasonably close to what one would expect of independent operations, since the probability of seeing a fresh value in the first read is about 33% and the probability of seeing a stale value in the second read is about 67%. The Pearson correlation between the outcomes of two successive reads is 0.0281, which is very low, and we conclude that eventual consistent reads are independent from each other.

Table 2: Successive Eventual Consistent Reads

First Read \ Second Read	Stale	Fresh
Stale	39.94% (189,926)	21.08% (100,1949)
Fresh	23.36% (111,118)	15.63% (74,337)

2.1.2 *Accessing from Multi Threads and Processes*

In the previous results, all read and write requests originate from the same thread. We did measurements for four other configurations:

1. A writer and a reader run in different threads in the same process,
2. A writer and a reader run in different processes on the same virtual machine in the same geographic domain as the data storage (US West),
3. A writer and a reader run on different virtual machines in US West, or
4. A writer and a reader run on different virtual machines, one in US West and one in Ireland.

In the first two cases, read and write requests are originated from the same IP address. In the third case, requests are originated from different IP addresses but from the same

geographical region. In the last case, requests are originated from different IP addresses in different regions.

Each experiment was run for 11 days as well. In all four cases the probability of reading updated values shows a similar distribution as in Figure 2. We conclude that consumers of SimpleDB see the same data consistency model regardless of where and how clients are placed.

2.2 Amazon S3

A similar measurement study was conducted on Amazon S3 for 11 days. In S3, storage consists of objects within buckets, so our writer updates an object in a bucket with the current timestamp as its new value, and each reader reads the object. In this experiment, we did measurements for the same five configurations as SimpleDB's case, i.e., a write and a reader run in a single thread, different threads, different processes, different VMs or different regions. Amazon S3 two types of write operations: *standard* and *reduced redundancy*. A standard write operation stores an object so that its probability of durability is at least 99.999999999%, while a reduced redundancy write aims at giving at least 99.99% probability of durability. The same set of measurements was performed with standard write and reduced redundancy write.

Documentation states that Amazon S3 buckets provide eventual consistency for overwrite put operations [4]; however, no stale data was ever observed in our study regardless of write redundancy options. It seems that staleness and inconsistency might be visible to a consumer of Amazon S3 only in executions such that there is a failure in the particular nodes of platform where the data is stored, during the time of their access; this seems a very low probability event.

2.3 Azure Table and Blob Storage

The experiment was also conducted on Windows Azure table and blob storages for eight days. Since it is not possible to start more than one process on a single VM (Web Role in this experiment), we did measurements for four configurations: a write and a reader run in a single thread, different threads, different VMs or different regions. On Azure table storage a writer updates a property of a table and a reader reads the same property. On Windows blob storage a write updates a blob and a reader reads it.

The measurement study observed no stale data at all. It is known that all types of Windows Azure storages support strong data consistency [24] and our study confirms it.

2.4 Google App Engine Datastore

Similar to SimpleDB, Google App Engine (GAE) datastore keeps key-accessed entities with properties, and it offers two options for reading: strong consistent read and eventual consistent read. However, the behavior we observed for eventual consistent read in GAE datastore is completely different from that of SimpleDB. It is known that the eventual consistent read of GAE datastore reads from a secondary replica only when a primary replica is unavailable [18]. Therefore, it is expected that consumer programmers see consistent data in most reads, regardless of the consistency option they choose.

We ran our benchmark application coded in Java and deployed in GAE. In GAE applications are not allowed to create threads; a thread automatically starts upon an HTTP request and it can run no more than 30 seconds. Therefore, each measurement on GAE runs for 27 seconds and we run measurements every 10 minutes for 12 days. The same set of measurements was performed with strong consistent read and eventual consistent read. Also, GAE offers no option to control the geographical location of applications. Therefore, we did measurements for two configurations: a writer and a reader run in the same application (i.e., thread), or a writer and a reader run in different applications. Each measurement consists of 9.4 writes and 2787.9 reads on average, and in total 3,727,798 reads and 12,791 writes happened on average for each configuration.

With strong consistent read no stale value was observed. With eventual consistent read and both roles in the same application, no stale value was observed. However 11 out of 3,311,081 readings ($3.3E^{-4}$%) observed stale values when a writer and an eventual consistent reader are run in different applications. We cannot conclude for certain whether stale values might sometimes be observed when a writer and a reader run in the same application; however, it suggests the possibility that GAE offers read-your-writes eventual consistency. In any case, consistency errors are very rare.

3. CONSISTENCY MODEL OF SIMPLEDB

We do not have a public description of the implementation approach used by SimpleDB. However our data from Section 2 shows that eventual consistent read of SimpleDB does not support monotonic reads or read-your-writes. This section investigates in more detail, what the consumer can determine about the data consistency model of SimpleDB eventual consistent read. Section 3.1 investigate the support of monotonic write consistency and Section 3.2 investigates the consistency among multiple data elements.

This section discusses the observations obtained in various studies; we speculate in Section 4.3 on mechanisms that might lead to these observations.

3.1 Monotonic Write Consistency

Vogels [32] has advocated the importance of the "monotonic write" property, because programming is notoriously hard if this is missing. The monotonic write property guarantees to serialize the writes by one process. It is not clear whether a consumer can test this, through looking at the values received in reads. To gain some insight, we ran a similar benchmark to the one which is described in Section 2, except it has only one thread which performs 100 repetitions of a small *cycle*, each of which updates a data element twice in a row and then reads the element repeatedly for three seconds; each read is placed in a bucket depending on the time interval from the start of the cycle till the read is submitted. We ran a measurement once every hour for nine days, and aggregated all the buckets from a given time after the cycle starts.

We refer to the value in the element before the cycle starts as `v0`, the value placed there in the first write as `v1` and then `v2` is written immediately afterwards. Figure 4 shows the probability of reading `v0`, `v1` or `v2` against the time from the start of the cycle. The total probability of reading `v0`, `v1` or `v2` at times between 50ms to 400ms are 0.097%, 66.339% and 33.564%, respectively. It appears that the second write is enough to ensure that no replica any longer contains the value from before the cycle started, even though we are still well below the period of 500ms from the first write, which our earlier data showed was the time till that write would be

visible in all replicas. We say that the second write "flushes" the first write to consistency.

When we modify the experiment to write three different values `v1`, `v2` and `v3` consecutively, the probability of reading `v0`, `v1`, `v2` or `v3` is 0.051%, 0.028%, 66.650% and 33.272%, respectively. Again, each write except the last seems to have been flushed, and this measurement suggests that each replica almost always contains a value that is no more than one write behind the latest.

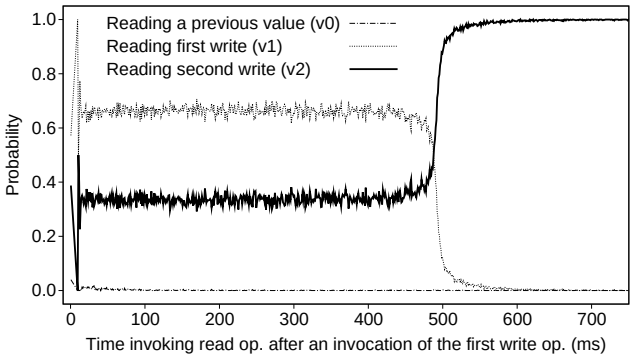


Figure 4: Consecutive Writes to One Item

Another study shows even more complexity in the consistency model that the consumer sees for SimpleDB, because flushing behavior varies depending on the content being written. This is just like the previous experiment, except that the same value is written in both writes of a cycle, that is `v1` = `v2`. Figure 5 shows the probability of reading `v0` or `v1` over time. The total probability of reading `v0` or `v1`, through the period between 50ms to 400ms after the write, is 66.526% and 33.474%, respectively. Note that this is quite different from what one would see if one just combined the curves for `v1` and `v2` in Figure 4. In particular, when `v1` = `v2`, after the second write some replicas clearly continue to hold the value from before the cycle, which is not so when `v1` ≠ `v2`. This suggests that the second write is ignored, and does not cause a flush of previous writes, if `v1` = `v2`.

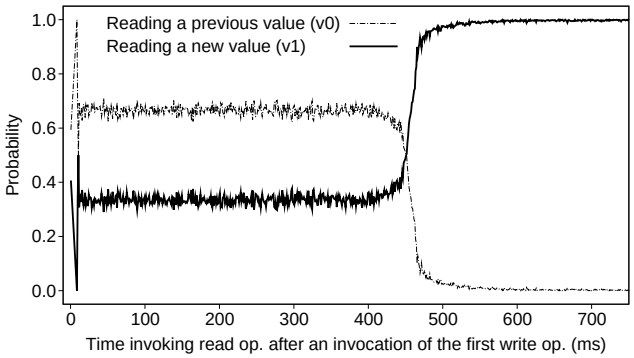


Figure 5: Writing the Same Value Twice

The SimpleDB data model is comprised of domains, items, attributes and values [5]. A domain is a set of items. An item is a set of attribute-value pairs. The write operation of SimpleDB can update multiple attribute-value pairs in an item at once. We use this fact to explore more closely the cases when a second write flushes the value in an earlier write. Our experiment is just like the previous one, except that the first write puts `v1` in two attributes, `A1` and `A2`, at once, and the second write puts `v2` in only `A1`. As in the previous experiments, one experiment writes different values, `v1` ≠ `v2`, in the first and second writes. The other experiment writes the same value, `v1` = `v2`, in both writes.

Figure 6 shows the probability of reading `v1` or `v2` when `v1` ≠ `v2`. The probability of reading a previous value (`v0`) is not zero; however, they are very small (less than 0.1%) and not shown in this figure. `A1` shows similar probability to the one shown in Figure 4. However, `A2` shows the complexity in the consistency model. Although `A2` is updated only once, by the first write, the probability of reading `v1` is very close to 100%. It indicates that the second write (i.e., writing `v2` on `A1`) affects the data consistency of `A2`, flushing the earlier write, despite the fact that it does nothing in `A2`.

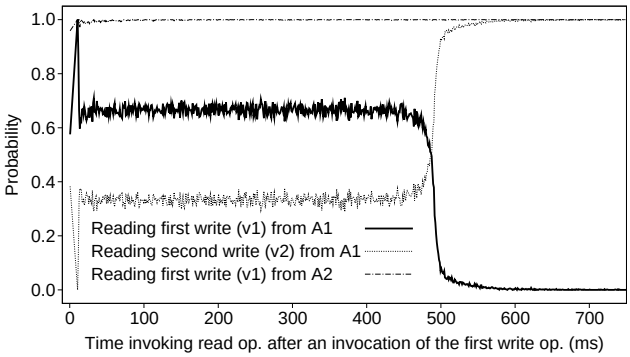


Figure 6: Writing in Two Attributes then in One

When `v1` = `v2`, the probability of reading `v1` from `A1` and that of reading `v1` from `A2` draw similar curves as the one in Figure 5. This suggests that the second write is ignored as it is redundant, delivering a subset of the information in the first write.

3.2 Inter-Element Consistency

NoSQL database systems usually provide limited support of transactions; in particular, there is typically avoidance of two-phase commit, and so the platform will typically not allow transactional update to elements that might be stored on different physical nodes. We explore this by having a writer thread modify two data elements (placing the current timestamp in each), and each reader examines those two locations. The SimpleDB data model is comprised of domains, items, attributes and values. We explore the effect of how closely the elements are related in the data model: they might be two attributes within one item, or attributes that are in different items within the same domain, or they might be in different items in different domains.

SimpleDB provides several operations to write and read values from elements; we also explore the effect of using various combinations of these to do writing and reading.

- `PutAttributes` updates attribute-value pairs in a certain item.
- `BatchPutAttributes` performs multiple `PutAttribute` operations in a single call.
- `GetAttributes` returns all attribute-value pairs in a certain item.
- `Select` returns a set of attribute-value pairs in a certain domain that match a query statment.

Table 3 shows the probability observed under different ways to write/read the values in two attribute-value pairs `X`

Table 3: Write and Read Two Elements in SimpleDB

		two `GetAttributes`				one `GetAttributes`				one `Select`			
		A (%)	B (%)	C (%)	D (%)	A (%)	B (%)	C (%)	D (%)	A (%)	B (%)	C (%)	D (%)
Values in same item	two `PutAttributes`	34.459	65.142	0.138	0.262	33.593	66.246	0.000	0.161	33.559	63.808	0.000	2.633
	one `PutAttributes`	12.050	21.820	22.554	43.576	33.859	0.000	0.000	66.141	33.756	0.000	0.000	66.244
	one `BatchPutAttributes`	11.789	21.964	22.507	43.740	33.649	0.000	0.000	66.351	33.610	0.000	0.000	66.390
Values in diff items in a domain	two `PutAttributes`	34.443	65.138	0.149	0.271	N/A	N/A	N/A	N/A	32.908	66.832	0.000	0.260
	one `BatchPutAttributes`	11.872	21.918	22.471	43.738	N/A	N/A	N/A	N/A	33.763	0.000	0.000	66.237
Values in diff domains	two `PutAttributes`	14.097	24.882	20.740	40.282	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	two `BatchPutAttributes`	14.187	24.924	20.740	40.149	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

and `Y`. N/A means this combination is infeasible; for example, it is impossible to read two values from different domains by one `GetAttributes` call. For each approach we show 4 probabilities: A for when the values of `X` and `Y` are both fresh, B when `X` is fresh but the value of `Y` is stale, C for the case where the value of `X` is stale but the value of `Y` is fresh, and finally D when both `X` and `Y` are stale. We run a measurement once every hour for seven days and obtained the total probability for reads that occur from 0ms to 450ms after the time of a write.

There are three distinct patterns observed. The first pattern is that the probabilities of A, B, C and D are about 12%, 22%, 22% and 44%, respectively. The second pattern is that the probabilities are about 34%, 0%, 0% and 66%. The third pattern is that the probabilities are about 34%, 66%, 0% and 0%.

The first pattern is what one would expect given independence between the items, based on the 33% probability for a single read seeing a fresh value. For example, when updating two attribute-value pairs with a single call of `PutAttributes` or `BatchPutAttributes` and then reading them with two consecutive calls of `GetAttributes`, the probability of reading fresh `X` and `Y` is about 12% (close to 33% $\times$ 33%).

The second pattern shows interaction between the items; both `X` and `Y` are stale or both are fresh. For example, it is observed when updates are done with one `PutAttributes`, and reading with a single call of `GetAttributes` or `Select`. The third pattern holds when two operations are used to do the writing; no matter how the reading is done, we see here that the first item has almost 100% chance of freshness, and the second has about 34% chance of freshness. This is the same phenomenon observed in Figure 6 with data no more than one write behind the current value.

4. TRADE-OFF ANALYSIS OF SIMPLEDB

The previous sections show the behavior of SimpleDB for different read options. For the platform provider, there are added costs for stronger consistency options (less availability, higher latency) [1]. We wish to see however what the consumer experiences, as this is what will guide the users of SimpleDB make a well-informed decision about which consistency option to ask for when reading.

We used the benchmark architecture described in Section 2. The measurement ran between 1 and 25 virtual machines in US West to write and read one attribute (which is a 14bytes string data) from an item in SimpleDB. Each virtual machine runs 100 threads, i.e., emulated clients, each of which executes one read or write request every second in a synchronous manner. Thus, if all requests' response time is below 1,000ms, the throughput of SimpleDB can be reported as 100% of the potential load. Three different read-write ratios were studied: 99% read and 1% write, 75% read and 25% write, and 50% read and 50% write cases. We run a measurement, which runs for five minutes with a set number of virtual machines, once every hour for one day.

4.1 Response Time and Throughput

The benefit of eventual consistent read in SimpleDB is explained as follows [5].

> The eventually consistent read option maximizes your read performance (in terms of low latency and high throughput).
>
> Since a consistent read can potentially incur higher latency and lower read throughput it is best to use it only when an application scenario mandates that a read operation absolutely needs to read all writes that received a successful response prior to that read.

To test this advice, we investigated the difference in response time, throughput and availability of the two options, as offered load increased. Figure 7 shows the average, 95 percentile and 99.9 percentile response time of eventual consistent reads and consistent reads at various levels of load. The result is obtained from the case of 99% read ratio and all failed requests are excluded. The result shows no visible difference in average response time; however, consistent read slightly *outperforms* eventual consistent read in 95 percentile and 99.9 percentile response time.

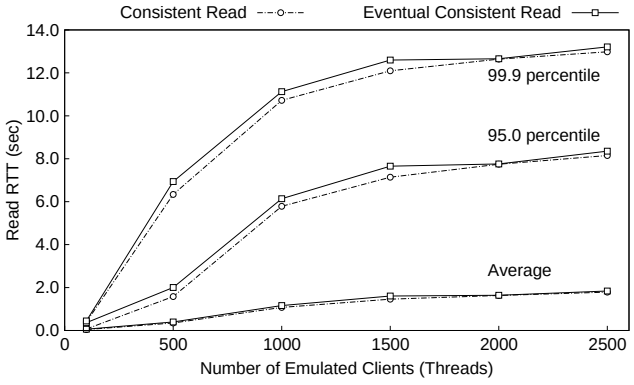


Figure 7: Response Time of Read on SimpleDB

Figure 8 and 9 show the average response time of reads and writes at various read-write ratios, plotted against the number of emulated clients. We conclude that changing the level of update-intensity does not have a marked impact.

Figure 10 shows the absolute throughput, the average number of processed requests per second. We also place whiskers surrounding each average with the corresponding

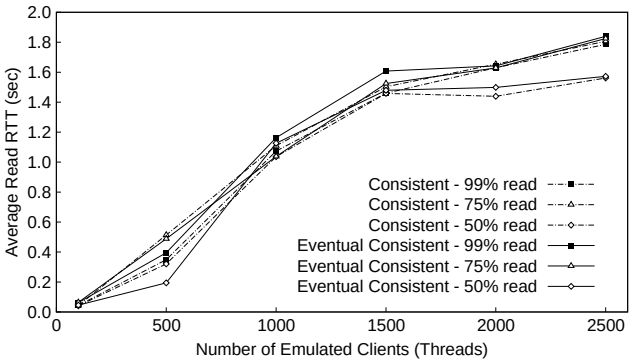


Figure 8: Response Time of Read on SimpleDB

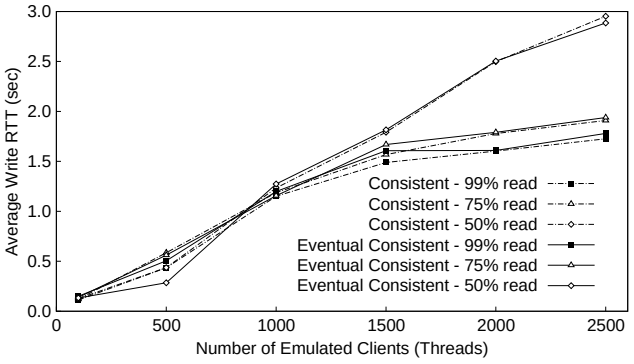


Figure 9: Response Time of Write on SimpleDB

minimum and maximum throughput. Similar to what we saw for response time, the result that consistent read slightly outperforms eventual consistent read, though the difference is not significant. Figure 11 shows the throughput as a percentage of what is possible with this number of clients. As the response time increased, each client sent less than one request every second and, therefore, the throughput percentage decreased.

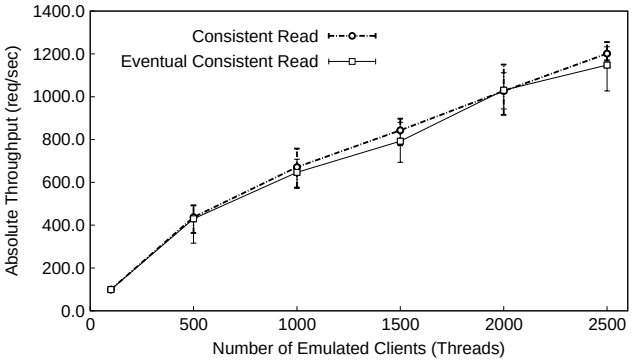


Figure 10: Processed Requests of SimpleDB

We observed that SimpleDB often returns exceptions (with status code 503: "Service is currently unavailable") under heavy load. Figure 12 shows the average failure rates of eventual consistent reads and consistent reads; each data point has whiskers to the corresponding maximum and minimum failure rates. Clearly the failure rate increased as offered load increased, but again we find that eventual consistent read does less well than consistent read, though the difference is not significant.

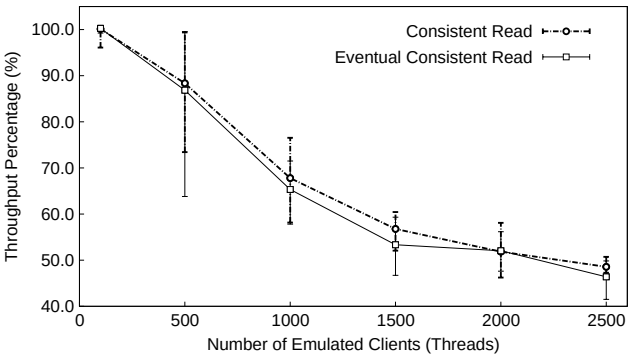


Figure 11: Throughput Percentage of SimpleDB

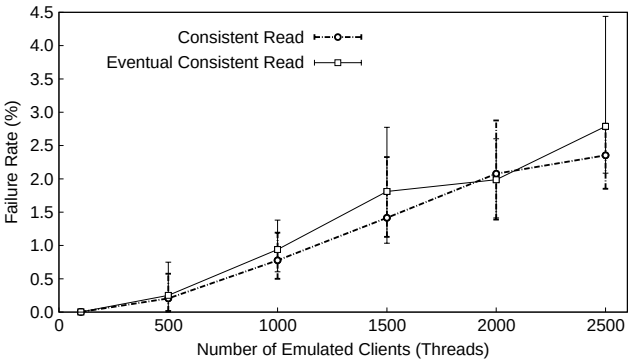


Figure 12: Request Failure Rate of SimpleDB

4.2 Financial Cost

Another way people sometimes view the consistency choice in cloud platforms is as a trade-off against financial cost [22]. In US West region, SimpleDB charges $0.154 per SimpleDB machine hour, which is the amount of SimpleDB's server capacity used to complete requests, and which can vary depending on factors such as operation types and the amount of data to access. We compared the financial cost of two read consistency options for the runs described above; Amazon reports the SimpleDB machine hour usage, so one can calculate the financial charge incurred for each request. The cost of read operations is constant, at $1.436 per 10^6 requests, regardless of the consistency options or workload. Also, the cost of write operations is constant at $3.387 per 10^6 requests as well.

4.3 Implementation Ideas

While our study takes a consumer view of the storage, we have ideas about the implementation based on our experiments. It seems feasible that SimpleDB maintains each item stored in 3 replicas, one primary and two secondaries. We suspect that an eventually consistent read chooses one replica at random, and returns the value found there, while a consistent read will return the value from the primary. This aligns with our experiments showing the same latency and computational effort for the two kinds of read. We are told (James Hamilton, personal communication) that an update is sent *synchronously* to all replicas. We conjecture that the update is applied to the data immediately at the primary replica, but that it remains buffered at the secondaries for a while, explaining the 66% probability of seeing a stale value in an eventual consistent read. Perhaps a write that

has been buffered at a replica is applied immediately when any subsequent write operation arrives (even one that is for a different data element), or when a timeout expires (usually 500ms after the write itself). This would explain the experiments of section 3.1 and 3.2, if we assume that all items within one domain are replicated at the same physical nodes, and items in different domains are replicated elsewhere. Further, maybe the system has an optimization that detects redundant write operations, and suppresses them. This must be sophisticated enough to detect not only repeated put requests, but also cases where one put is merely a subset of the previous update. We also suggest, from Table 3, that when multiple attributes are modified within a single operation such as `PutAttributes` or `BatchPutAttributes`, the activity happens with a single message, since in these cases we do not see any immediate application of the writes, but we do see that a following read always observes the same freshness status for each attribute.

5. CAN CONSUMERS RELY ON OUR RESULTS?

Our paper reports on the properties and performance of various cloud-based NoSQL storage platforms, as we observed them during some experiments. A natural concern is whether our results can be extrapolated to predict what the consumer will experience when using one of the platforms. We really can't say!

All the usual caveats of benchmarks measurements apply to us. For example, the workload may be unrepresentative for the consumer's needs, perhaps because in our tests the size of the writes is so small, and the number of data elements is small. Similarly, the metrics quoted may not be what matters to the consumer, the consumer's staff may be more or less skilled in operating the system than we were, perhaps the experiments were not run for long enough and the figures might reflect chance rather than system fundamentals, etc.

As well, there is a particular issue when measuring cloud systems: the vendor might change any aspect of hardware or software without notice to the consumer. For example, even if the algorithm used by a platform currently provides read-your-writes, the vendor could shift to a different implementation that lacked this guarantee. As another example, a vendor that currently places all replicas within a single data center might implement geographical distribution, with replicas stored across data centers for better reliability. Such a change could happen without notice to the consumers, but it might lead to a situation where eventual consistent reads have observably better performance than consistent reads. Similarly, the background load on the vendor's systems might have a large impact, on latency or availability or consistency, but the consumer cannot control or even measure what that load is at any time [29]. For all these reasons, our observations that eventual consistent reads are no better for the consumer, might not hold in the future.

The observations reported in this paper were mainly obtained in October and November, 2011. We had conducted similar experiments in May, 2011. Most aspects were similar between the two sets of experiments, in particular the 500ms latency till SimpleDB reached 99% chance for a fresh response to a read, the high chance of fresh data in eventual consistent reads in S3, Azure and GAE, and the lack of performance difference between SimpleDB for reads with different consistency. Other aspects had changed, for example in the earlier measurements there was less variation in the response time seen by reads on SimpleDB.

6. RELATED WORK

A broad survey of database replication techniques is given in [21].

Many papers have described particular architectures and algorithms for storage in the cloud. These owe much to earlier designs for distributed and especially mobile systems. The concept of eventual consistency arose in work on disconnected operation [13]. Saito and Shapiro offer a valuable survey of techniques that keep replicas loosely synchronized, such as those that provide eventual consistency [28]. Specifically dealing with the cloud, we note several papers from the past five years that describe particular systems: Yahoo!'s PNUTS [10], Amazon's Dynamo [12], Google's Bigtable [9]. The algorithms may be similar to those used in some of the consumer-accessible storage services.

Much research has investigated the performance and cost effectiveness of cloud computation platforms [20, 22, 30], using benchmark applications simulating typical web applications. For example Kossmann et al use the TPC-W workload with platforms that provide both storage and computation service, and report on throughput (accepted requests per second), financial cost per throughput achieved, and also the variability of the cost. In contrast, our paper focuses directly on NoSQL storage systems, and especially on their consistency properties.

Some previous papers have measured consistency aspects of storage platforms. For a single SQL-interface database engine, Fekete et al [14] define a benchmark that reports how often a consistency condition is violated. They observe rates that depend on the amount of contention between items, and the spacing of read and write operations within a transaction. Considering cloud platforms, Florescu and Kossman [16] argued for the importance of including consistency among the features that are measured, and they suggested that system evaluation should identify the tradeoff between consistency and other properties such as financial cost.

The CloudCmp [25] project benchmarks many features of cloud computing. It includes a measure of "time to consistency" of the storage layer. CloudCmp shows a very different pattern to what we found, and they indicate that the median time to consistency is only about 80 milliseconds. This seems to be because they report the delay from the write until the *first* time that a read returns the up-to-date value, whereas we note that such a read may be followed by others that show stale values; thus we measure the period till *almost all* reads see the recent write. Another difference, though probably not the reason for the different outcomes, is that CloudCmp does an insertion of a new key as the write operation, while we update the value in an existing element. Our work also goes further than CloudCmp by considering more aspects than just reading the recent write; we measure for example properties like monotonicity of reads and inter-element consistency.

A blog [26] reported results, like ours in Section 4, that eventual consistent and consistent reads have similar latency and throughput in SimpleDB. They did not explore the financial costs.

A different approach to measuring consistency of cloud

storage platforms is taken by Anderson et al [6], where they record lengthy traces with interleaved operations, and after the fact they check for cycles in various conflict graphs to determine whether various properties hold. The properties they analyse are those that are important in parallel hardware design, such as regular or safe registers, rather than the properties usual in cloud storage platforms such as eventual consistency with monotonic reads.

There is also work on formally defining weak consistency properties. Usually eventual consistency is defined in terms of internal properties such as the state of the replicas, but Fekete and Ramamritham [15] have proposed a definition based only on the results that are returned to the consumer. Extra properties such as session properties, that can strengthen the programmability of eventual consistency, were identified by Terry et al [31]. Awareness of these was spread by the important advocacy of Vogels [32]. Aiyer et al [3] define "consistability" based on the percentage of the operating period in which different consistency models are present.

Kraska et al. [23] consider having a layer above the storage, where different consistency models are supported with different performance properties, and then the client can choose dynamically what is appropriate. They build a theoretical model to analyze the impact of the choice of consistency model in terms of performance and cost, and propose a framework that allows for specifying different consistency guarantees on data. The results discussed in our paper could be used as inputs, i.e., actual behavior, performance and cost of different consistency models, to complement Kraska's work.

In contrast to the NoSQL databases we have studied, Microsoft Azure SQL[5] aims to support the traditional relational model and transactional guarantees in the cloud. It provides strong consistency in reads. However, it has a restriction on the size of the data (a database can grow up to 50GB) and does not support distributed transactions.

7. CONCLUSION

To achieve high availability and low latency, many cloud data storage platforms (or particular operations within a platform) use techniques that avoid two-phase commit and/or synchronous access to a quorum of sites. Thus they can't guarantee strong consistency. It is commonly said that developers should program around this by designing applications that can work with eventual consistency or similar weak models. We have examined the experience of the consumer of cloud storage, in regard to weak consistency and possible performance tradeoffs to justify it. This information should help a developer who is seeking to understand the properties of the new NoSQL storage platforms for the cloud, and who needs to make sensible choices about which storage platform to use.

We found that platforms differed widely in how much weak consistency is seen by consumers. On some platforms, we found that the consumer did not observe any inconsistency or stale data, over several million reads through a week. While inconsistency is presumably possible, it seems very rare; perhaps only happening if there is a failure of one of the nodes or communication links actually used in the computation. Since replication of storage is typically done on 3 or at most 4 nodes, such a failure is unlikely during the computation. Here the risks from inconsistency seem less important compared to other sources of data corruption, such as bad data entry, operator error, customers repeating input, fraud by insiders, etc. Any system design needs to have recourse to manual processes to fix the mistakes and errors from these other sources, and the same processes should be able to cover rare inconsistency-induced difficulties. On these platforms, we wonder whether the developer might sensibly choose to treat eventual consistent reads as if they are consistent, and accept the rare errors as part of doing business.

On Amazon SimpleDB, the consumer who requests eventual consistent reads experiences frequent stale reads and inter-item inconsistency. Also, this choice does not provide other desirable properties like read-your-writes and monotonic reads. Thus the programmer who uses eventual consistent reads must take great care in application design, to code around the dangers of this. However, we found no compensating benefit to the programmer: no reduction in latency, increase in observed availability or lower financial cost, for eventual consistent reads compared to using consistent reads (which are also offered as an option in SimpleDB). There may be benefits to the platform provider when eventual consistent reads are done, but at present these gains seem not to be passed on to the consumer. Thus on this platform in its current implementation, we see no reason for a developer to code with eventual consistent reads.

This work highlights the importance of a research agenda to expand the scope of the service level agreement between cloud provider and customer, to describe more carefully and quantitatively the consistency properties that a platform offers. Tool support for monitoring service levels should also include reporting on consistency aspects. We plan to pursue these ideas.

[5]www.microsoft.com/windowsazure/sqlazure/

Acknowledgement

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. We thanks Sherif Sakr and Shirley Goldrei for proofreading.

8. REFERENCES

[1] D. Abadi. Problems with CAP, and Yahoo's little known NoSQL System. `dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html`. [Accessed: Oct 1, 2010].

[2] R. Agrawal et al. The Claremont Report on Database Research. *SIGMOD Record*, 37(3):9–19, 2008.

[3] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *Usenix Workshop on Hot Topics in Systems Dependability (HotDep'08)*, 2008.

[4] Amazon Web Services. S3 FAQs. `aws.amazon.com/s3/faqs/`. [Accessed: July 6, 2010].

[5] Amazon Web Services. SimpleDB FAQs. `aws.amazon.com/simpledb/faqs/`. [Accessed: July 6, 2010].

[6] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store

actually provide? In *Usenix Workshop on Hot Topics in Systems Dependability (HotDep'10)*, 2010.

[7] M. Armbrust et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California, Berkeley, Feb 2009.

[8] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[9] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Sym. on Operating Systems Design and Implementation*, 2006.

[10] B. F. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. In *Proc Very Large Databases (VLDB'08)*, pages 1277–1288, 2008.

[11] M. Creeger. Cloud Computing: An Overview. *ACM Queue*, 7(5):3–4, 2009.

[12] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-Value Store. *Operating Systems Review*, 41(6):205–220, 2007.

[13] A. J. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proc ACM Conference on Principles of Distributed Computing (PODC'87)*, pages 1–12, 1987.

[14] A. Fekete, S. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. In *Proc Very Large Databases (VLDB'09)*, pages 467–478, 2009.

[15] A. Fekete and K. Ramamritham. Consistency models for replicated data. In *Replication (LNCS 5959)*, pages 1–17. Springer Verlag, 2010.

[16] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[17] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.

[18] Google. Datastore Python API Overview. `code.google.com/appengine/docs/python/datastore/overview.html`. [Accessed: Jul 22, 2010].

[19] J. M. Hellerstein. Datalog redux: experience and conjecture. In *Proc. ACM Principles of Database Systems (PODS'10)*, pages 1–2, 2010.

[20] Z. Hill, M. Mao, J. Li, A. Ruiz-Alvarez, and M. Humphrey. Early Observations on the Performance of Windows Azure. In *AMC Workshop on Scientific Cloud Computing*, June 2010.

[21] B. Kemme, R. Jiménez-Peris, and M. Patiño Martínez. *Database Replication (Synthesis Lectures on Data Management 7)*. Morgan and Claypool, 2010.

[22] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *ACM International Conference on Management of Data*, pages 579–590. ACM, 2010.

[23] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *International Conference on Very Large Data Bases*, August 2009.

[24] S. Krishnan. *Programming Windows Azure: Programming the Microsoft Cloud*. O'Reilly, 2010.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Internet Measurement Conference*, page to appear, 2010.

[26] H. Liu. The cost of eventual consistency. `http://huanliu.wordpress.com/2010/03/03/%EF%BB%BFthe-cost-of-eventual-consistency/`. [Accessed Oct 12, 2010].

[27] S. Malkowski et al. Empirical Analysis of Database Server Scalability using an N-tier Benchmark with Read-Intensive Workload. In *ACM Symposium on Applied Computing*, pages 1680–1687. ACM, 2010.

[28] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[29] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *Proc Very Large Databases (VLDB'10)*, pages 460–471, 2010.

[30] W. Sobel et al. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *Workshop on Cloud Computing and its Applications*, October 2008.

[31] D. B. Terry et al. Session guarantees for weakly consistent replicated data. In *Proc of International Conference on Parallel and Distributed Information Systems (PDIS'94).*, pages 140–149. IEEE Computer Society, 1994.

[32] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[33] A. Weiss. Computing in the Clouds. *netWorker*, 11(4):16–25, 2007.

Using Data for Systemic Financial Risk Management

Mark Flood
University of Maryland
mdflood@rhsmith.umd.edu

H. V. Jagadish
University of Michigan
jag@umich.edu

Albert Kyle
University of Maryland
akyle@rhsmith.umd.edu

Frank Olken
LBL
frankolken@acm.org

Louiqa Raschid
University of Maryland
lraschid@umiacs.umd.edu

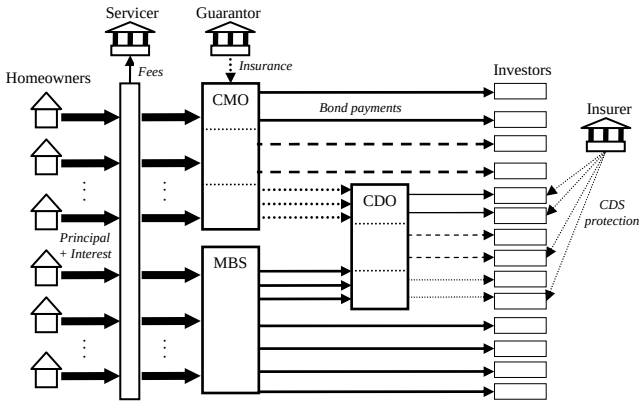
ABSTRACT

The recent financial collapse has laid bare the inadequacies of the information infrastructure supporting the US financial system. Technical challenges around large-scale data systems interact with significant economic forces involving innovation, transparency, confidentiality, complexity, and organizational change, to create a very difficult problem. The post-crisis reform legislation has created a unique opportunity to rebuild financial risk management on a solid foundation of information management principles. This should help reduce operating costs and operational risk. More importantly, it will support both the monitoring and the containment of financial risk on a previously unprecedented scale. These objectives will pose several information management challenges, including issues of knowledge representation, information quality, data integration, and presentation. This paper presents a vision of an information-rich financial risk management system, and a research agenda to facilitate its realization.

1. INTRODUCTION

The banking crisis that erupted in 2008 underscored the need for reductions in systemic financial risk. [4] While there were myriad interacting causes, a central theme in the crisis was the proliferation of complex financial products that overwhelmed the system's capacity for appropriately diligent analysis of the risks involved. In many cases, the information available about products and counterparties was minimal. In response, the Dodd-Frank Wall Street Reform Act – among its many regulatory changes – has created an Office of Financial Research (OFR) with the mandate to establish a sound data management infrastructure for systemic-risk monitoring. The OFR will contain a Data Center (OFR/DC) to manage data for the new agency.

For many years, both financial firms and their regulators have been hampered by a state of "data anarchy," despite (or perhaps because of) the enormous volumes of mission-critical data the industry handles daily. The widespread use of PCs has dispersed access, ownership, and control of data throughout the firm, to create multiple, overlapping data silos. The result is disparate, inconsistent and inaccurate information. Furthermore, cross-institutional barriers often inhibited regulators from obtaining the data that could have resulted in recognizing systemic risk early on, and thus, potentially preventing a timely response to emergent failures.



To be more concrete, consider the highly stylized example of home mortgage payments passing through the securitization chain depicted in the figure above. Homeowners submit principal and interest payments to a servicing bank, which collects a fee, passing the bulk on to securitization pools (trusts) that own the mortgages. A "pass-through" mortgage-backed security (MBS) pro-rates the payments to its bondholders, thus providing diversification benefits. A collateralized mortgage obligation (CMO) does the same, but structures the payments into prioritized tranches targeted to specific credit-risk and maturity clienteles (dashed lines indicate contingent cash flows). MBSs and CMOs typically have some credit-support, here from a third-party guarantor. Some of the MBS and CMO bonds are held directly by investors, but others are pooled into a collateralized debt obligation (CDO), which re-tranches the cash flows again to focus the credit and maturity exposures further. In the figure, some of the CDO investors have purchased additional credit protection, in the form of credit default swaps (CDS).

This greatly simplified depiction elides any number of important intricacies, such as the loan origination process, fixed vs. floating interest rates, homeowners' prepayment and curtailment options, the choice of funding sources to hedge interest-rate risk, tax and accounting treatment, government guarantees, the role of ratings agencies, portfolio management for the CDO pool, etc., etc. Nonetheless, the figure is already quite complicated. As most of us know, each mortgage loan is itself a complex legal contract,

5th Biennial Conference on Innovative Database Systems Research (CIDR 2011): January 9-12, 2011, Asilomar, California, USA.

but the securitization (MBS, CMO and CDO) agreements are typically much more involved still, with prospectuses and other offering documents that run on for hundreds of pages of legalese.

For an investor or investment manager, a key problem is determining the nature and magnitude of the financial risks she is exposed to through her various contracts. [3] A special challenge for the regulator is to determine the risks to the financial system as a whole, taking into account the correlations and dependencies between defaults, prepayments, movements in interest rates and other prices, institutional leverage, liquidity shocks, etc. Achieving the vision of computing risk through multiple counter-parties to complex contracts presents significant information management challenges. [5] In Section 2 we provide some background on the financial information system, and in Section 3 we discuss the information management challenges.

2. FINANCIAL SYSTEMS BACKGROUND

Systemic risk monitoring is inherently complex. Financial institutions acquire information from hundreds of sources, including prospectuses, term sheets, corporate filings, tender offers, proxy statements, research reports and corporate actions. They load the data into master files and databases providing access to prices, rates, descriptive data, identifiers, classifications, and credit information. They use this information to derive yields, valuations, variances, trends and correlations. They feed raw data streams and derived data into pricing models, calculation engines and analytical processes. These data are linked to accounting, trade execution, clearing, settlement, valuation, portfolio management analysts, regulators and market authorities. Further complicating these daunting technical challenges, there are also strong incentives for many market participants to restrict transparency around risks [7].

The data quality gap in finance is an evolutionary outcome of years of mergers and internal realignments, exacerbated by business silos and inflexible IT architectures. Difficulties in unraveling and reconnecting systems, processes, and organizations – while maintaining continuity of business – have made the problem intractable. Instead, data are typically managed on an ad-hoc, manual and reactive basis. Workflow is ill defined, and data reside in unconnected databases and spreadsheets with multiple formats and inconsistent definitions. Integration remains point-to-point and occurs tactically in response to emergencies. Many firms still lack an executive owner of data content and have no governance structure to address funding challenges, organizational alignment or battles over priorities.

Financial risk and information managers are gradually recognizing the concepts of metadata management, precise data definitions based on ontologies, and semantic models and knowledge representation as essential strategic objectives. [1] These same concerns apply with equal urgency to the financial regulators tasked with understanding individual firms and the overall system. A sound data infrastructure and open standards are necessary, both for effective regulation and for coordinating industry efforts. The history of patchwork standards and partial implementations demonstrates the tremendous obstacles to consensus over shared standards in the absence of a disinterested central authority.

The CDO depicted in the figure provides a simple example of the scale of the problem. A CDO might pool bonds from scores of MBSs, each of which pools hundreds of mortgages, entailing a total of many thousands of pages of contractual language. All of this legalese must be implemented in computer and information systems for each of the hundreds of participants in the pipeline.

Systemic risk monitoring is not a precise science, but there is a general consensus that it should consider at least [6]:

- forward-looking risk sensitivities to stressful events (e.g., what would a 1% rise in yields mean for my portfolio?);
- margins, leverage, and capital for individual participants (e.g., how large a liquidity shock could I absorb before defaulting?);
- the contractual interconnectedness of investors and firms (e.g., if Lehman Bros. fails, how will that propagate to me?)
- concentration of exposures, relative to market liquidity (e.g., how many banks are deeply exposed to California real estate?)

The OFR/DC will need the following types of information [2]:

- Financial instrument reference data: information on the legal and contractual structure of financial instruments, such as prospectuses or master agreements, including data about the issuing entity and its adjustments based on corporate actions;
- Legal entity reference data: identifying and descriptive information, such as legal names and charter types, for financial entities that participate in financial transactions, or that are otherwise referenced in financial instruments;
- Positions and transactions data: terms and conditions for both new contracts (transactions) and the accumulated financial exposure on an entity's books (positions);
- Prices and related data: transaction prices and related data used in the valuation of positions, development of models and scenarios, and the measurement of micro-prudential and macro-prudential exposures.

Together, these data can resolve the fundamental questions of who (i.e., which specific legal entity) is obligated to pay how much to whom, on which future dates, and under what contingencies. Based on this, one can assess both firm-wide and system-wide risk, and gains insights on the risks to consumers posed by particular financial products and practices.

3. RESEARCH CHALLENGES

To determine systemic risk from the large volume of complex and heterogeneous data describing the financial system, regulators (and industry participants) must understand ownership hierarchies, and counterparty and supply-chain relationships. They must keep up with financial innovation, corporate actions, and micro- and macro-level events occurring continually among thousands of entities around the world. New regulations will expose additional data for analysis. All of this must be analyzed and distilled to measures of systemic risk. This section briefly discusses the associated data management research challenges, including issues of knowledge representation, data integration, information quality, metadata and change management, data presentation, security, privacy and trust.

3.1 Data Representation and Complex Models

The notion of *risk* is central in finance, and it poses some fundamental questions. Where is risk intrinsically? Does it pertain to an analytical model or to the instrument itself? How is systemic risk best represented? Risk often varies with time, and its evaluation typically is based on multiple sources of time-varying data. How should one best ensure the timeliness of risk evaluation and also indicate staleness of data sources, given that there are many of these?

Many data sources will have risk(s) associated with them. As one merges multiple data sources and computes derived information, these notions of risk are propagated. For example, a bank may track a measure of default risk for each mortgage it writes. These individual risk numbers are informative, but are even more valuable when aggregated to the level of the full portfolio. Risk measures may also be combined across risk dimensions (e.g., borrower creditworthiness and geographic concentration) to provide a richer risk picture. Unfortunately, aggregating risk measures is usually more difficult than simply adding them up – considerable additional information is needed. The challenge is to derive minimal risk representations with enough information for downstream derivation. What information is needed may depend on the risk models used, which in turn may depend on what data are available, creating a chicken-and-egg problem.

Putting risk aside, financial information systems must have an adequate representation for many complex entities. For example, formulae themselves (e.g., a rule for calculating payouts under particular contingencies) should often be treated as data. It is straightforward to treat a formula as a simple string of text characters. Ideally, this formula-as-data would have more sophisticated handling, so that parts of formulae can be recognized, queried, and even computed. Actually manipulating formulae would be useful, but this may require more self-modification than most current database systems allow.

Another complex entity of interest is the accounting system. Even formal financial reporting rules frequently allow significant discretion in how positions and activities are treated, leading to large discrepancies in reported values. For example, internal transfer pricing schemes work to report profits in a firm's lowest-tax jurisdiction. Working with these issues requires at least that the accounting system be indicated as metadata. Queries on accounting system used are likely, and metadata query facilities may be needed.

Automated reasoning with complex contracts requires that the contracts be stated in a machine manipulable form. It appears that current knowledge representation techniques may be able to get us close to where we need to be in this regard.

3.2 Data Integration

Financial information management and data sharing confront the standard problems of data integration one would expect, given the multiplicity and heterogeneity of data sources. Data integration has been extensively studied, with many partial solutions already in place, and much progress over the past several decades. We believe financial systems are yet another important context and motivation for this line of work.

While the OFR/DC may have regulatory powers to force some standardization across sources, we nonetheless expect considerable heterogeneity. For example, regulators now have broad fiat authority to require fixed tags for certain data types, but an unsettled research question is which data should be tagged. Unless there is a standard ontology providing shared definitions and semantics, comparing financial data across multiple institutions will continue to be a challenge.

Consider primary keys or identifiers (such as CUSIP codes for North American securities). Currently, there are multiple competing numbering schemes in many markets; in some cases, a single identifier might even be reused for several instruments. Other markets may have very limited identifier coverage; for example, a CDO owner in the figure above would likely have trouble identifying all of the specific mortgages underlying his security. The research need is to: (a) determine which objects should have identifiers; (b) specify techniques to track identifiers across contractual netting and novation, and across corporate mergers and separations; (c) cross-reference different identifier standards; and (d) include checksums in identifiers to catch data entry errors. A further challenge is to define a protocol for the evolution of identifiers. Financial data sharing at the instance level may be simple on the one hand because much of the data appears as numeric streams. However, without precise agreement on the definition of terms or formulae used, comparing simple numeric values or other features of the data without access to metadata may be meaningless, or introduce confusion and error. Also, the OFR cannot collect everything, so triage based on the usefulness of the data is needed.

3.3 Data Quality

There are at least three distinct reasons for poor data quality in financial systems: incompleteness or error in the source(s) of data; errors in data integration; and fraud. We deal with each in turn.

One might expect some data sources, such as trade data, to be reasonably complete. However, "trade breaks" (i.e., cancelled transactions) due to un-reconcilable discrepancies in transaction details are painfully common. Others data sources, such as company data, are naturally incomplete or subject to interpretation. Yet other data represent estimates of aggregates, such as macroeconomic data. It may be possible to characterize the incompleteness and possible error in many data sources, but it is an open question how to record and reflect this in downstream computation. Furthermore, data quality may be measured and corrected at different levels, including the application level.

Given the large number and the variety of data sources, errors in data integration are to be expected. It is likely that integration will occur on an automated, best-efforts basis, with human correction applied to fix some, but probably not all of the errors. A research issue is to characterize aspects of the integration process most likely to affect derived results, so that scarce human effort can be devoted to checking the most critical areas.

There are strong incentives for fraud in financial systems, and many individual firms currently use fraud detection software. Integrated data from multiple sources should increase the opportunities to detect fraud, through comparison and reconciliation of discrepancies between data sources. Many large-scale frauds (e.g., the Madoff and Barings scandals) have required the entry of fictitious contracts into trading systems; since every contract has at least two counterparties, a simple check for the existence of the other side of the deal could have revealed the

crimes. There is also a need for an automated protocol when a problem is detected – often one may want additional proof of fraudulent activity to avoid alerting the fraudsters prematurely.

3.4 Streaming, Change Management and Performance

Many data sources (e.g., high-frequency trading) produce large volumes of data. Furthermore, in some instances, rapid response is required, the "flash crash" of May 2010 being an obvious example. Time-stamp granularity is a concern for fast moving phenomena. Streaming data techniques are likely needed.

Many data series are recorded and published right away, but this timeliness implies that many data sources will only show estimates when first made public. For example, government economic indicators are typically revised as new information is revealed, frequently with multiple restatements. When revisions are published, procedures should exist to trigger an update of derived data that were based on the original numbers.

3.5 Metadata Management: Ontologies, Open Standards and Provenance

Besides the important issues of accounting systems and model formulae, there is a host of other relevant metadata that must be recorded adequately, and folded into derivations where needed. For example, many historical series on corporate information should be merger-adjusted, just as equity prices must be adjusted for stock splits and dividends. In addition to metadata on what is measured, it is also important to track who is performing the measurement – and how – to understand the reliability of derived results. In other words, extensive provenance management is required. Banks today already use audit trails, and the technology to do this is the natural place from which to build a full-fledged provenance recording and management system.

3.6 Data Presentation

Even with all of the above technologies in place, systemic risk will not reduce to a single global number. This is equally true for many other derived results of interest. Rather, these results will at best be derived as a function of various model assumptions and inputs. In many cases, there may be no closed form derivation at all – rather all we may be able to do is to simulate under specified conditions and obtain results thereby. In other words, the decision-maker cannot be given a single number that is easy to understand. Rather, there is a range of numbers, with complex dependence on multiple factors. Under such circumstances, data presentation becomes very important. A poorly designed decision "dashboard" may be worse than having no standard presentation at all. Research is required into the most effective presentation of complex data and the results derived from them.

3.7 Security, Privacy and Trust

The need for security at the technical level and trust at the organizational level are keys to achieving the goals of the OFR. In addition to the expected challenges, the open sharing of metadata and ontologies may not always be possible due to perceived competitive advantage associated with such knowledge. Important parts of the financial industry are cloaked in secrecy. One challenge will to develop an appropriate set of property and privacy rights to delineate between public and confidential financial information. Another research issue will be the design of a physical and information security infrastructure for the OFR to maintain confidentiality where required.

4. CONCLUSIONS

This paper identifies some of the key reasons for data anarchy in the financial industry, including multiple heterogeneous silos, the data quality gap, lack of standards and the inherent complexity and uncertainty involved. In response to this and other shortcomings, the Dodd-Frank Wall Street Reform Act has created an Office of Financial Research (OFR) with a mandate to establish a sound data-management infrastructure for systemic-risk monitoring. The new OFR includes a Federal Financial Data Center to manage data for the new agency. Achieving acceptable and successful solutions for meeting risk monitoring objectives will present several information management challenges. These are briefly discussed here, and include knowledge representation, information quality, data integration, metadata management, change management, presentation, security, privacy and trust.

5. ACKNOWLEDGMENTS

A workshop on financial information management was partially supported by the National Science Foundation grant IIS1033927 and the Pew Charitable Trusts in July 2010. The material in this paper is derived from discussions at this workshop.

6. REFERENCES

[1] Bennett, M., 2010. *Enterprise Data Management Council Semantics Repository*. Internet resource (downloaded 26 Sep 2010). DOI= http://www.hypercube.co.uk/edmcouncil/.

[2] The Committee to Establish the National Institute of Finance (CE-NIF), 2009. *Data Requirements and Feasibility for Systemic Risk Oversight*. Technical report (30 Nov 2009). DOI= http://www.ce-nif.org/images/docs/ce-nif-generated/nif_datarequirementsandfeasibility_final.pdf.

[3] Duffie, D., 2010. The Failure Mechanics of Dealer Banks, *J. of Econ. Perspectives*. 24, 1 (Winter, 2010) 51-72. DOI= http://pubs.aeaweb.org/doi/pdfplus/10.1257/jep.24.1.51.

[4] Engle, R. and Weidman, S. 2010. *Technical Capabilities Necessary for Systemic Risk Regulation: Summary of a Workshop*. (Washington DC, USA, Nov. 3, 2009). National Academies Press, Washington DC. DOI= http://www.nap.edu/catalog.php?record_id=12841.

[5] Haldane, A. 2009. *Why Banks Failed the Stress Test*. Speech, Bank of England (9-10 February 2009). DOI= http://www.bankofengland.co.uk/publications/speeches/2009/speech374.pdf.

[6] Lo, Andrew W., 2009. Regulatory Reform in the Wake of the Financial Crisis of 2007–2008, *J. of Financial Econ. Policy*, 1, 1 (2009) 4-43. DOI= http://www.emeraldinsight.com/journals.htm?issn=1757-6385&volume=1&issue=1.

[7] Rowe, D., 2009. Fostering Opacity, *Risk Magazine* (July 2009). DOI= http://davidmrowe.com/riskmag.php.

Computational Journalism: A Call to Arms to Database Researchers

Sarah Cohen
School of Public Policy
Duke Univ.
sarah.cohen@duke.edu

Chengkai Li
Dept. of Comp. Sci. & Eng.
Univ. of Texas at Arlington
cli@uta.edu

Jun Yang
Dept. of Comp. Sci.
Duke Univ.
junyang@cs.duke.edu

Cong Yu
Google Inc.
congy@umich.edu

1. INTRODUCTION

The digital age has brought sweeping changes to the news media. While online consumption of news is on the rise, fewer people today read newspapers. Newspaper advertising revenues fell by a total of 23% in 2007 and 2008, and tumbled 26% more in 2009 [1]. This continuing decline of the traditional news media affects not only how news are *disseminated* and *consumed*, but also how much and what types of news are *produced*, which have profound impact on the well-being of our society. In the past, we have come to rely heavily upon the traditional news organizations for their investigative reporting to hold governments, corporations, and powerful individuals accountable to our society. The decline of traditional media has led to dwindling support for this style of journalism, which is considered as cost-intensive and having little revenue-generating potential.

Today, there are fewer reporters gathering original material than in a generation. By some reports, full-time newsroom employment has fallen by one-quarter over the past ten years [2]. Bloggers, citizen journalists, and some non-profit news agencies have made up for only a small part of this loss. The growth in the online news organizations has been mostly in the role of *aggregators*, who read other blogs and news reports, and select, aggregate, edit and comment on their findings. There is a real danger that the proud tradition of original, in-depth investigative reporting will fade away with the ailing traditional news media.

Luckily, a second trend is on our side: the continuing advances in computing. We are connecting people together on unprecedented scales. Data collection, management, and analysis have become ever more efficient, scalable, and sophisticated. The amount of data available to the public in a digital form has surged. Problems of increasing size and complexity are being tackled by computation. Could computing technology—which has played no small part in the decline of the traditional news media—turn out to be a savior of journalism's watchdog tradition?

In the summer of 2009, a group (including two authors of this paper) of journalists, civic hackers, and researchers in social science and computer science gathered for a workshop at Stanford on the nascent field of *computational journalism*, and discussed how computation can help lower cost, increase effectiveness, and encourage participation for investigative journalism. In this paper, we present a more focused perspective from database researchers by outlining a vision for a system to support mass collaboration of investigative journalists and concerned citizens. We discuss several features of the system as a sample of interesting database research challenges. We argue that computational journalism is a rich field worthy of attention from the database community, from both intellectual and social perspectives.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

2. A CLOUD FOR THE CROWD

News organizations today have little time and resources for investigative pieces. It is economically infeasible for most news organizations to provide their own support for investigative journalism at a sufficient level. To help, we envision a system based on *a cloud for the crowd*, which combines computational resources as well as human expertise to support more efficient and effective investigative journalism.

The "cloud" part of our vision is easy to understand. Treating computing as a utility, the emerging paradigm of cloud computing enables users to "rent" infrastructure and services as needed and only pay for actual usage. Thus, participating news units can share system setup and maintenance costs, and each reporter can access a much bigger pool of resources than otherwise possible. Popular tools, such as those supporting the Map/Reduce model, have made scalable data processing easy in the cloud. These tools are a perfect fit for many computational journalism tasks that are inherently data-parallel, such as converting audio or scanned documents to text, natural language processing, extracting entities and relationships, etc. Finally, sharing of infrastructure and services encourages sharing of data, results, and computational tools, thereby facilitating collaboration.

Cloud for computational journalism is becoming a reality. A pioneering example is DocumentCloud.org, started by a group of journalists at ProPublica and the *New York Times* in 2008. It hosts original and user-annotated documents as well as tools for processing and publishing them. Conveniently, it uses a Ruby-based Map/Reduce implementation to perform document OCR on Amazon EZ2. Going beyond DocumentCloud's document-centricity, the system we envision would also help manage, integrate, and analyze *structured data*, which may be either extracted from text or published by a growing number of public or government sources. With structured data, we can draw upon a wealth of proven ideas and techniques from databases, ranging from declarative languages, continuous querying, to parallel query processing. Implementing them in the cloud setting raises new challenges, and is a direction actively pursued by the database community. Computational journalism may well be a "killer app" for this line of research.

We would like to emphasize the "crowd" part of our vision more, however. While the "cloud" part copes with the need for

computational resources, investigative journalism will never be fully automated by computation. How can we leverage the "crowd" to cope with the need for human expertise? As a start, DocumentCloud allows sharing of human annotations on documents and facilitates collaborative development of tools. With declarative information extraction and declarative querying, we could further share and reuse results (final or intermediate) of data processing tasks initiated by different users. From the perspective of the system, such collaboration occurs opportunistically. Ideally, we want to *actively* direct the efforts of the crowd. For example, in 2009, *The Guardian* of London put up close to half a million pages of expense documents filed by British MPs on the Web, and asked viewers to help identify suspicious items. Within 80 hours, 170,000 documents were examined, making this crowdsourcing effort a spectacular success [3]. As an example from the database community, the *Cimple* project on community information management [4] relies on user feedback to verify and improve accuracy of information extracted from Web pages. Many existing approaches assign jobs to the crowd in simple ways (e.g., pick any document to check). They will certainly get the job done in the long run, but if we have in mind an idea for a story with a tight deadline—which is often the case for journalists—assigning many jobs whose outcomes bear little relevance to our goal will dilute the crowd's efforts.

What we envision instead is a system that intelligently plans (or helps to plan) the crowd's efforts in a *goal-driven* fashion. Given a computational task (e.g., a query), the system would generate a tentative result based on the data it currently has. At this point, the result can be quite uncertain, because the data contain all sorts of uncertainty, ranging from imperfect information extraction to errors in publicly released datasets. Suppose the result looks newsworthy, and a journalist is allocated a limited amount of time and resources to investigate this lead. To leverage the power of the crowd, our system would come up with mini-tasks to be crowdsourced. These mini-tasks can be as simple as checking an entity-relationship extracted from a document, or as complex as reconciling entries from two different public databases. Different lists of mini-tasks would be presented to different users according to their expertise and preference. Mini-tasks whose completion contributes most to reducing the overall result uncertainty will be listed as having a higher priority. As results of mini-tasks become available, the system would reevaluate the overall result, and adjust and reprioritize the remaining mini-tasks accordingly. This idea can be seen as a generalization of the pay-as-you-go approach to data integration in *dataspaces* [5], which considered mini-tasks that establish correspondences between entities from different data sources.

To better illustrate the new challenges and opportunities involved, put yourself in the shoes of a journalist who just noticed a huge number of blog posts about high crime rates around the Los Angeles City Hall. First, are there really that many posts about high crime rates in this area, or did the automated extraction procedure pick up something bogus? Second, does having a large number of blog posts necessarily increase the credibility of the claim, or did most of these posts simply copy from others? Knowing that the number of *original* sources for a story is almost always very low, a seasoned journalist will likely start with a few popular posts, verify that they indeed talk about high crime rates around Los Angeles City Hall, and then trace these posts back to find the original sources. When planning for crowdsourcing, our system should try to mimic the thought process of seasoned journalists. In particular, it would be suboptimal to assign mini-tasks for verifying extraction results from a lot of posts, because the number of posts making a claim is a poor indicator for the accuracy of the claim anyway, and checking just a few may boost our confidence in the extraction procedure enough. It is also suboptimal to ask the crowd to trace the sources of many posts, because a handful of them may lead us to the few original sources.

In this case, it came down to a couple of sources: a Los Angeles Police Department site for tracking crimes near specific addresses, and EveryBlock.com, which publishes large bodies of public-domain data (such as crimes and accidents) by location and neighborhood. Further investigation reveals that EveryBlock.com republished data from LAPD, so our task reduces to that of verifying the claim in the LAPD database (a topic that we shall return to in Section 3). Interestingly, according to the geocoded map locations of crimes in the database, the numbers check out: the crime rate at 90012, ZIP code for the Los Angeles City Hall, indeed ranked consistently as the highest in the city. But a true investigative journalist does not stop here; in fact, there is where the fun begins. It would be nice for our system to help journalists quickly eliminate other possibilities and zoom in on the fun part.

As it turned out, there was a glitch in the geocoding software used by the LAPD site to automatically convert street addresses to map locations. Whenever the conversion failed, the software used the default map location for Los Angeles, right near the City Hall, hence resulting in a disproportionally high crime rate. Arriving at this conclusion does require considerable skill and insight, but our system can help by providing easy access to the full dataset (with street addresses included), and by crowdsourcing the mini-tasks of checking crime records that have been geocoded to the default location (if no alternative geocoding software is available).

Much of what we described in this example took place in real life, and was the subject of a 2009 story in the *Los Angeles Times* [6]. We do not know how many bloggers picked up the false information, but the fact that the *Los Angeles Times* published this piece about the software glitch instead of a column on crimes near the City Hall is both comforting and instructive. As the Internet has made it trivial to publish (and republish) information—especially with the proliferation of social networking—there is a real danger of misinformation going viral. It is important for computational journalism to help preserve journalistic principles and to facilitate fact-checking (more on these in Section 3).

Many research challenges lie ahead of us in supporting intelligent planning of crowdsourcing for investigative journalism. Before we can hope to replicate or improve the cost-benefit analysis implicitly carried out in the mind of a seasoned journalist, we first need frameworks for representing prior knowledge and uncertainty in data (raw and derived) and reasoning with them. There has been a lot of work on probabilistic databases [7], and it would be great to put the techniques to a serious test. For example, how do we represent a large directed acyclic graph of uncertain dependencies among original sources and derived stories? How do we capture the belief that the number of original sources is small? We believe studying our application will necessitate advances in data uncertainty research.

Given a declarative specification of what we seek from data, we also need methods to determine what underlying data matter most to the result. Akin to sensitivity analysis, these methods are crucial in prioritizing mini-tasks. Work on lineage [8] took an

important initial step towards this direction, but we are interested in not only *whether* something contributes to the result, but also *how much* it would affect the result when it turns out to be something else. Work on dataspaces [5] laid out an interesting direction based on the concept of the *value of perfect information*, but with general mini-tasks and more complex workflows involving extraction, cleansing, and querying, the problem becomes more challenging.

In addition to the benefit of a mini-task, we will also need to quantify its cost. The idea of exploring the cost-benefit tradeoff has been investigated in the context of *acquisitional query processing* in sensor networks [9]. The human dimension of the crowd creates many new problems. Interests, expertise, and availability vary greatly across users. Some mini-tasks may never be picked up. Sometimes users do a poor job. Therefore, it is difficult to predict a mini-task's cost and result quality (which affects its *actual* benefit). Also challenging are the problems of adjusting crowdsourcing plans dynamically based on feedback, coordinating crowdsourcing among concurrent investigations, and allocating incentives using, say, the Amazon Mechanical Turk. Efforts such as American Public Media's Public Insight Network have taken a qualitative approach toward building and using participant profiles. It would be interesting to see whether a more quantitative approach can be made to work for a crowd.

To recap, we envision a system for computational journalism based on "a cloud for crowd," which hosts tools and data (raw and derived, unstructured and structured), runs computational tasks, and intelligently plans and manages crowdsourcing. It combines resources and efforts to tackle large tasks, and it seeks to leverage and augment human expertise. Within the system, there are endless possibilities for innovative applications of computing to journalism. In the next section, we describe a few specific ideas with clear database research challenges, which help attract participation and maintain a healthy ecosystem that encourages accountability in both subjects and practice of reporting.

3. FROM FINDING ANSWERS TO FINDING QUESTIONS

Much of the database research to date has focused on answering questions. For journalism, however, finding interesting questions to ask is often more important. How do we define interestingness? Where do we come up with interesting questions? To gain some insights, we start with two news excerpts as examples:

The water at American beaches was seriously polluted … with the number of closing and advisory days at ocean, bay and Great Lakes beaches reaching more than 20,000 for the fourth consecutive year, according to the 19th annual beach water quality report released today by the Natural Resources Defense Council (NRDC). … [10]

… Hopkins County also maintained the lowest monthly jobless rate in the immediate eight-county region for the 29th consecutive month. … [11]

Like the two excerpts above, many news stories contain factual statements citing statistics that highlight their newsworthiness or support their claims. Oftentimes, these statements are essentially English descriptions of queries and answers over structured datasets in the public domain.

What if, for each such statement in news stories, we get a pointer to the relevant data source as well as a query (say, in SQL) whose answer over the data source would support the statement?

With such information, we can turn stories *live*. We are used to static stories that are valid at particular points in time. But now, our system can continuously evaluate the query as the data source is updated, and alert us when its answer changes. In the beach water quality example, a continuous query would be able to monitor the alarming condition automatically year after year. If more detailed (e.g., daily) data are available, we can tweak the query to make monitoring more proactive—instead of waiting for an annual report, it would alert us as soon as the number of closing days this year has reached the threshold. Thus, the query in the original story lives on, and serves as a lead for follow-ups.

We will be able to make stories *multiply*. The original story may choose to focus on a particular time, location, entity, or way of looking at data. But given the data source and the query, we can generalize the query as a parameterized template, and try other instantiations of it on the data to see if they lead to other stories. In the jobless rate example, an interested user may instantiate another query to compare her own county of residence against its neighbors. Note that the original query requires more skills than might appear at first glance: it searches for a Pareto-optimal point (x, y) to report, where x is the number of neighboring counties and y is the number of consecutive months. By enabling a story to multiply, we facilitate reuse of investigative efforts devoted to the story, thereby alleviating the lack of expertise, especially at smaller local news organizations.

We will be able to *fact-check* stories quickly. Fact-checking exposes misinformation by politicians, corporations, and special-interest groups, and guards against errors and shady practices in reporting. Consider the following example from FactCheckED.org [12], a project of the Annenberg Public Policy Center. During a Republican presidential candidates' debate in 2007, Rudy Giuliani claimed that adoptions went up 65 to 70 percent in the New York City when he was the mayor. The city's Administration for Children's Services (ACS), established by Giuliani in 1996, made a similar claim by comparing the total number of adoptions during 1996-2001 to that during 1990-1995.

If we were given the data source and the query associated with the claim above, our system can simply run the query and compare its result against the claim. You may be surprised (or perhaps not so) to find that many claims exposed by FactCheckED.org cannot even pass such simple checks. This example, however, requires more effort. According to FactCheckED.org, the underlying adoption data, when broken down by year, actually show that adoption began to slow down in 1998, a trend that continued through 2006. Lumping data together into the periods of 1990-1995 and 1996-2001 masks this trend.

Even when simple automatic fact-checking fails, making sources and queries available goes a long way in helping readers uncover subtle issues such as the one above. When reading stories (or research papers), we often find ourselves wondering why authors have chosen to show data in a particular way, and wishing that we get to ask questions differently. In reality, most of us rarely fact-check because of its high overhead—we need to identify the data sources, learn their schema, and write queries from scratch. By making sources and queries available for investigation, we can significantly increase the crowd's participation in fact-checking to help us ensure accountability.

We have discussed three useful tools—making stories live, making stories multiply, and fact-checking stories—all based on the assumption of having sources and queries to support claims. Naturally, the next question is how to get such information. We could ask makers of claims to provide this information (akin to

requiring data and methods for scientific papers), but this approach does not always work. These people may no longer be available, they may have obtained the answers by hand, or they may have motives to withhold that information. Instead, can our system help us identify the data source and reverse-engineer the query associated with a claim?

This problem seems to entail solving the natural language querying problem, which several research and productization efforts attempted in the past but has not caught on in practice. We believe, however, that two new approaches will give us extra leverage. First, we have the text not only for the query, but also for its answer. Evaluating a candidate query on a candidate dataset and comparing the answer can serve as a powerful confirmation. Second, and perhaps more importantly, as more people use our tools on stories, our system can build up, over time, a library containing a wealth of information about data sources, queries and answers, as well as how they are used in actual stories. Suggestions for relevant data sources may come from stories on similar topics. Reverse engineering of queries can benefit from seeing how similar texts have been translated.

This library also leads us to the interesting possibility of building a *reporter's black box*. An investigative piece may involve hundreds of hand-crafted queries on a structured database. Apprentices of investigative journalism face a steep learning curve to write interesting queries. In fact, the majority of these queries seem to conform to some standard patterns—grouping, aggregation, ranking, looking for outliers, checking for missing values, etc. A reporter's black box will be a tool that automatically runs all sensible instantiations of "standard" query templates on a database. For databases that are updated, the black box will automatically monitor them by evaluating the queries in a continuous fashion. The library of datasets and queries maintained by our system will help us discover and maintain collections of interesting query templates. It will also help us find patterns across datasets (or those with particular schema elements), allowing us to "seed" template collections for new datasets.

The number of interesting query templates for a dataset may be large, and the number of instantiations will be even larger, since a parameter can potentially take on any value in the dataset. When the reporter's black box runs on a dataset, it should present query-answer pairs in order of their newsworthiness, which helps journalists focus their efforts. Ranking criteria may incorporate generic ones such as query length (compact queries are more compelling) or template-specific ones such as answer robustness (answers that change with small perturbations to query parameters are less compelling). The library of datasets and queries maintained by our system is also useful. For example, queries with templates that have been used by many high-impact stories probably should be ranked higher, especially if their answers have not appeared in old stories based on the same templates.

Running a large number of queries and monitoring tasks *en masse* poses interesting system and algorithmic challenges. Cloud parallelization helps. Techniques in multi-query optimization and scalable continuous query processing are applicable. However, queries in the reporter's black box can get quite complex (if you have trouble motivating skyline queries, look here), which complicates shared processing. On the other hand, more sharing arises from the fact that many queries are instantiated from the same template. The need to produce ranked query-answer pairs also presents unique challenges and opportunities. Instead of devoting an equal amount of computational resources to each query, we would give priority to queries that are more likely to yield high-ranking query-answer pairs.

Interestingly, there is one area of news where a specialized reporter's black box has been immensely successful—sports. It is amazing how commentaries of the form "player X is the second since year Y to record, as a reserve, at least α points, β rebounds, γ assists, and δ blocks in a game" can be generated seemingly instantaneously. Replicating this success for investigative journalism is difficult. While sports statistics attract tremendous interests and money, public interest journalism remains cash-strapped, and has to deal with a wider range of domains, smarter "adversaries," more diverse and less accurate data sources, and larger data volumes. The ideas presented in this section will hopefully help combat these challenges, by providing efficient, easy-to-use computational tools that aid journalists and citizens in their collaboration to ensure accountability, and by creating a positive feedback cycle where participation helps improve the effectiveness of these tools.

4. CONCLUSION

In this short paper, we have outlined our vision for a system to support collaborative investigative journalism. We have focused on several features of the system to highlight a few important database research challenges. As the Chinese saying goes, we are "throwing a brick to attract a jade"—there are many more interesting problems, both inside and outside the realm of database research: privacy, trust and authority, data mining, information retrieval, speech and vision, visualization, etc.

The need to realize this vision is already apparent. With the movement towards accountability and transparency, the amount of data available to the public is ever increasing. But at the same time, the ability to work with data for public interest journalism remains limited to a small number of reporters. Computation may be the key to bridge this divide and to preserve journalism's watchdog tradition. We hope to motivate you, both intellectually and civically, to join us in working on computational journalism.

Acknowledgments *The authors would like to thank all participants of the 2009 workshop on* Accountability through Algorithm: Developing the Field of Computational Journalism*, organized by James T. Hamilton and Fred Turner, and hosted by the Center for Advanced Study in the Behavioral Sciences at Stanford.*

C.L. and J.Y. would also like to thank HP Labs in Beijing, China for hosting their visits in the summer of 2010, during which some ideas in this paper were discussed.

5. REFERENCES

[1] Pew Research Center's Project for Excellence in Journalism. "The state of the news media." March 15, 2010.

[2] American Society of News Editors. "Decline in newsroom jobs slows." April 11, 2010.

[3] Andersen. "Four crowdsourcing lessons from the Guardian's (spectacular) expenses-scandal experiment." Nieman Journalism Lab. June 23, 2009.

[4] Doan et al. "Community information management." *IEEE Data Engineering Bulletin*, 29(1), 2006.

[5] Jeffrey, Franklin, and Halevy. "Pay-as-you-go user feedback for dataspace systems." *SIGMOD* 2008.

[6] Welsh and Smith. "Highest crime rate in L.A.? No, just an LAPD map glitch." *The Los Angeles Times*. April 5, 2009.

[7] Dalvi, Ré, and Suciu. "Probabilistic databases: diamonds in the dirt." *CACM*, vol. 52, 2009.

[8] Widom. "Trio: a system for data, uncertainty, and lineage." In Aggarwal, editor, *Managing and Mining Uncertain Data*, Springer, 2009.

[9] Madden et al. "TinyDB: an acquisitional query processing system for sensor networks." *TODS*, 30(1), 2005.

[10] Natural Resources Defense Council. "Beach closing days nationwide top 20,000 for fourth consecutive year." July 29, 2009.

[11] Bruce Alsobrook. "Local unemployment rate best in eight-county area for 29th straight month." *The Sulphur Springs News-Telegram*. September 22, 2010.

[12] FactCheckED.org. "Dubious adoption data." June 6, 2007.

Ibis: A Provenance Manager for Multi-Layer Systems

Christopher Olston and Anish Das Sarma
Yahoo! Research

ABSTRACT

End-to-end data processing environments are often comprised of several independently-developed (sub-)systems, e.g. for engineering, organizational or historical reasons. Unfortunately this situation harms usability. For one thing, systems created independently tend to have disparate capabilities in terms of what metadata is retained and how it can be queried. If something goes wrong it can be very difficult to trace execution histories across the various sub-systems.

One solution is to ship each sub-system's metadata to a central metadata manager that integrates it and offers a powerful and uniform query interface. This paper describes a metadata manager we are building, called *Ibis*. Perhaps the greatest challenge in this context is dealing with data provenance queries in the presence of mixed granularities of metadata—e.g. rows vs. column groups vs. tables; map-reduce job slices vs. relational operators—supplied by different sub-systems. The central contribution of our work is a formal model of multi-granularity data provenance relationships, and a corresponding query language. We illustrate the simplicity and power of our query language via several real-world-inspired examples. We have implemented all of the functionality described in this paper.

1. INTRODUCTION

Modern systems are often comprised of multiple semi-independent sub-systems. Examples at Yahoo come in at least three varieties:

- **Stacked:** systems with higher-level abstractions stacked upon lower-level systems, e.g. Oozie [2] stacked on Pig [3] stacked on Hadoop [1].
- **Pipelined:** data flows through a sequence of systems, e.g. a system for ingesting RSS feeds, then a system for processing the feeds, then a system for indexing and serving the feeds via a search interface.
- **Side-by-side:** Two systems serving the same role might operate side-by-side during a migration period, with responsibility being transferred to a replacement system gradually, to allow the new system to be vetted and fine-tuned. In another scenario, redundant systems are deployed in a permanent side-by-side configuration, with each one targeting a different point in some performance tradeoff space such as latency vs. throughput.[1]

Modularity in these forms facilitates the creation of complex systems, but can complicate operational issues, including monitoring and debugging of end-to-end data processing flows. To follow a single RSS feed from beginning to end may require interacting with half a dozen sub-systems, each of which likely has different metadata and different ways of querying it.

Yahoo architects would like to reduce the manual effort required to track data across sub-systems. Solutions that rely on standardization efforts or deep code modifications are undesirable, and in fact unrealistic when using third-party components, or even in-house ones that are already mature and widely deployed.

Motivated by this challenge, we are creating *Ibis*, a service that collects, integrates, and makes queryable the metadata produced by different sub-systems in a data processing environment. This approach has three main advantages:

- Users are provided with an integrated view of metadata, via a uniform query interface.
- Boilerplate code for storing and accessing metadata is factored out of n data processing sub-systems, into one place (Ibis). Moreover, since Ibis specializes in metadata management it will likely do a better job, versus the data processing sub-systems for which metadata management falls into the "bells and whistles" category.
- The lifespan of the metadata is decoupled from that of the data to which it refers, and even from the lifespans of the various data processing sub-systems.

1.1 Provenance Metadata Heterogeneity

Arguably the most complex type of metadata to manage is *data provenance*, which is the focus of this paper. A system that aims to integrate provenance metadata from multiple sub-systems must deal with nonuniformity and incompleteness.

[1]For example, one might find a low-latency/low-throughput feed processing engine deployed side-by-side with a high-latency/high-throughput engine, with time-sensitive feeds (e.g. news) handled by the former and the majority of feeds handled by the latter.

To begin with, different sub-systems often represent data and processing elements at different granularities. Data granularities range from tables (coarse-grained) to individual cells of tables (fine-grained), with multiple possible mid-granularity options, e.g. rows vs. columns vs. temporal versions. Process descriptions also run the gamut from coarse-grained (e.g. a SQL query or Pig script) to fine-grained (e.g. one Pig operator in one retry attempt of one map task), also with multiple ways to sub-divide mid-granularity elements (e.g. map and reduce phases vs. Pig operations (which may span phases) vs. parallel partitions).

Moreover, links among processing and data elements sometimes span granularities. For example, one system at Yahoo records a link from each (row, column group, version) combination (e.g. latest release date and opening theater for the movie "Inception") to an external source feed (e.g. Rotten Tomatoes).

Finally, one cannot assume that each sub-system gives a complete view of its metadata. At Yahoo, and presumably elsewhere, metadata recording is enhanced over time as new monitoring and debugging needs emerge. Recording "all" metadata at the finest possible granularity sometimes imposes unacceptable implementation and performance overheads on the system producing the metadata, as well as on the system capturing and storing it.

Ibis accommodates these forms of diversity and incompleteness with a *multi-granularity provenance model* coupled with query semantics based on an *open-world assumption* [9]. This paper describes Ibis's provenance model, query language and semantics, and gives examples of their usage. The model and language have been fully implemented on top of an ordinary RDBMS, using simple query rewriting techniques. Performance and scalability issues are subjects of ongoing work, and are not the focus of the present paper.

1.2 Outline

The remainder of this paper is structured as follows. We discuss related work in Section 2. Then we present Ibis's provenance data model in Section 3. The semantics and syntax of Ibis's query language are given in Sections 4–6. We describe a prototype implementation of a storage and query manager for Ibis in Section 7.

2. RELATED WORK

Provenance metadata management has been studied extensively in the database [5] and scientific workflow [7] literature, including the notion of offering provenance management as a first-class service, distinct from data and process management, e.g. [14]. Many aspects of our approach borrow from prior provenance work, and are somewhat standard at this point. For example, modeling provenance relationships as (source data node, process node, target data node) triples, use of free-form key/value attributes, and use of a declarative SQL/datalog-style query language, are all commonalities between Ibis and other approaches such as Kepler's COMAD provenance manager [4]. However, most prior work on provenance has focused on tracking a single system's provenance metadata, and consequently has generally assumed that provenance metadata is rather uniform, and/or can be tightly coupled to the data in one system.

We provide the first formal framework—data provenance model, query language and semantics—for integrated management of provenance metadata that spans a rich, multi-dimensional granularity hierarchy. The core contribution of our work is a set of rules for inferring provenance relationships across granularities. These inference rules have carefully-chosen, precise semantics and have been embedded in our query language and system.

Several prior projects offer (restricted) multi-granularity models, but none of them focus on formal semantics for inferring relationships when provenance is queried:

- Kepler's COMAD model [4] and ZOOM user views [6] deal with uni-dimensional granularity hierarchies in data (COMAD's nested collections) or process (ZOOM's sub-workflows), but neither supports multi-dimensional granularity hierarchies, a combination of data and process hierarchies, or the ability for queries to infer provenance relationships across granularities.
- The *open provenance model* [10] shares our goal of offering a generic framework for representing and accessing provenance metadata from diverse sources. The open provenance model aims to support multi-dimensional granularity hierarchies via the notion of "refinement," but it does not provide specific semantics for multi-granularity refinement, or formal rules for carrying provenance relationships across granularities in the data model, query language or system.
- References [8, 13] consider annotations on arbitrary two-dimensional sub-regions of relational tables, but do not deal with provenance linkage and inference.

One branch of the Harvard PASS project [11] shares our goal of managing provenance that spans system layers. That work restricts its attention to coarse-grained provenance, and tackles numerous issues around capturing and cleaning the provenance metadata (e.g. APIs, object naming schemes and cycle detection). It is complementary to the work we present in this paper, which considers multi-granularity provenance and focuses on how to represent and query it. As our project moves forward to tackle the capture and cleaning issues, we hope to leverage the PASS work.

Lastly, Ibis supports relatively simple forms of provenance—where-provenance and lineage ("flat" why-provenance)—which suffice for most use-cases we have encountered at Yahoo. More elaborate forms of provenance that associate logic expressions with provenance links (e.g. witness sets and how-provenance; see [5]) are not our focus.

3. DATA PROVENANCE MODEL

This section introduces Ibis's model of provenance graphs that connect data and process elements at various granularities.

3.1 Data and Process Granularities

An Ibis instance is configured with *granularity sets* (gsets) that describe the possible granularities of data and process elements and their containment relationships.

DEFINITION 3.1 (GSET). *A gset is defined by a bounded partially-ordered set (poset)* $G = (\mathbf{G}, \preceq, g_{\max}, g_{\min})$*, where* $\mathbf{G}$ *gives the finite set of granularities, "*$\preceq$*" denotes containment and defines a partial order over* $\mathbf{G}$*, and there exist unique maximal and minimal elements* $g_{\max}, g_{\min} \in \mathbf{G}$*, i.e.,* $\forall g \in \mathbf{G} : g_{\min} \preceq g \preceq g_{\max}$.

Figure 1 gives example data and process gsets, which are based on some of Yahoo's web data management applications. The arrows denote containment relationships: an

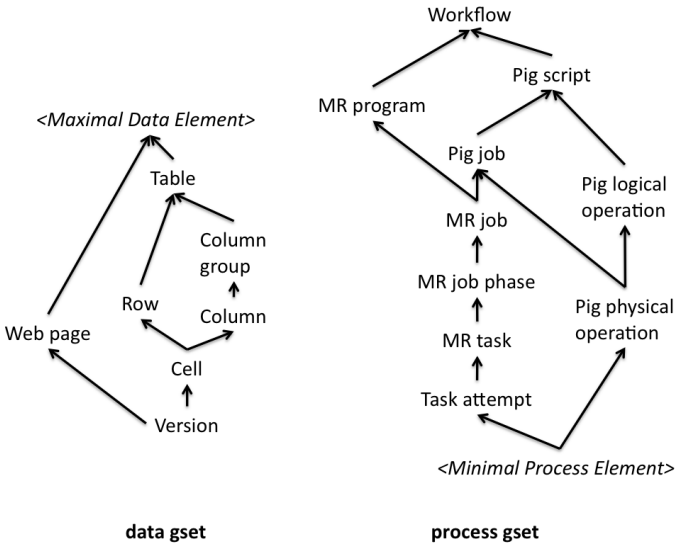


Figure 1: Example gsets.

arrow from X to Y denotes $X \preceq Y$. Intuitively, $X \preceq Y$ implies that each element at granularity Y must have an element at finer-granularity X, but the converse may not hold, i.e., each element at granularity X does not need to have an element at coarser-granularity Y.

In our example, data is either part of a relational table or a free-form web page. Relational tables are divided horizontally into rows, and vertically into column groups, which are further subdivided into columns. A row/column combination is a cell. A table cell can have multiple versions of data, e.g. reflecting multiple conflicting possible data values, or temporally changing values. Web pages also have versions, corresponding to multiple crawled snapshots.

Processing, at the coarsest granularity, is driven by workflows whose steps are either map-reduce programs or pig scripts. An execution of a program or script is called a job. Pig jobs are comprised of a series of map-reduce jobs, which are in turn broken into two phases (map and reduce). Each phase is partitioned into map or reduce tasks, which undergo one or more execution attempts. Syntactically, Pig scripts consist of sequences of logical operations. Pig logical operations are compiled into sequences of physical operations, which perform the work inside the map/reduce task attempts.[2]

When a new Ibis instance is configured, one data gset and one process gset must be supplied. If unique maximal and minimal elements are absent from either gset, Ibis creates them automatically (e.g. *<Maximal Data Element>* and *<Minimal Process Element>* in Figure 1).

3.2 Data and Process Elements

We start by defining *basic elements*, the atomic unit of a data or process item. Each basic element is specified by a particular granularity, a unique identifier, and parent (coarser-granularity) basic elements as defined below.

DEFINITION 3.2 (BASIC ELEMENT). *A basic element $b = (g, id, \mathcal{P})$ is defined by a granularity g in the data or process gset, a globally unique[3] id, and a set $\mathcal{P}$ of ids of basic elements that are direct parents in the containment hierarchy.*

Our next definition formalizes the notion of containment of basic elements:

DEFINITION 3.3 (BASIC ELEMENT CONTAINMENT). *Given two basic elements $b_1 = (g_1, id_1, \mathcal{P}_1)$ and $b_2 = (g_2, id_2, \mathcal{P}_2)$, b_1 contains b_2 iff either $id_1 \in \mathcal{P}_2$ or $\exists b^* \in \mathcal{P}_2$ such that b_1 contains b^* (according to recursive application of this definition).*

Intuitively, b_2 is contained in b_1 if b_1 is a direct parent (i.e., coarser granularity element) or an ancestor in the granularity hierarchy.

Next we define the notion of "granularizing" basic elements to the finest possible granularity, a concept that will be used later to infer new relationships among elements. Granularization simply consists of finding all basic elements of the finest granularity contained in a given element:

DEFINITION 3.4 (BASIC ELEMENT GRANULARIZATION). *Given basic element $b = (g, id, \mathcal{P})$ and minimal element $g_{\min}$ in the gset containing g, the* granularization *of b, written $\mathcal{G}(b)$, is $\{b' = (g_{\min}, id', \mathcal{P}') : b$ contains $b'\}$.*

Next we define the notions of complex element types and complex elements, which allow us to compose elements from multiple basic elements of different granularities.

DEFINITION 3.5 (COMPLEX ELEMENT TYPE). *A complex element type $T = \{g_1, g_2, \ldots, g_n\}$ is a set of granularities such that all members are from the same gset (i.e. all data granularities or all process granularities) and no two members $g_i, g_j \in T$ satisfy $g_i \preceq g_j$.*

An example complex element type is { row, column group }, which denotes a data element defined by the intersection of a particular row and a particular column group. Each complex element type has an associated *attribute set* $A = \{a_1, a_2, \ldots, a_m\}$, $m \geq 0$, e.g. { owner, storage location }.

DEFINITION 3.6 (COMPLEX ELEMENT). *A complex element $E = (id, T = \{g_1, g_2, \ldots, g_n\}, \{b_1, b_2, \ldots, b_n\})$ consists of a globally unique id, a type T, and a basic element b_i corresponding to each granularity $g_i \in T$.*

An example complex element is (8, { row, column group }, { row 5, column group 3 }).

Finally, we extend the definition of granularization to complex elements, in the natural way:

DEFINITION 3.7 (COMPLEX ELEMENT GRANULARIZATION). *Given complex element $E = (id, T, \{b_1, b_2, \ldots, b_n\})$, the granularization of E is $\mathcal{G}(E) = \bigcap_{1 \leq i \leq n} \mathcal{G}(b_i)$.*

3.3 Provenance Graph

Ibis manages a *provenance graph* that relates sets of complex elements to one another via three-way relationships. Figure 2 shows an example provenance graph from a simple web information extraction scenario in which movie data

[2]In general there is no containment relationship between Pig operations and map/reduce tasks, or even map/reduce phases (e.g. some join operations span phases).

[3]Our model can be extended easily to accommodate scoped ids, e.g. row ids that are unique within the scope of a table would be identified via (table id, row id) pairs.

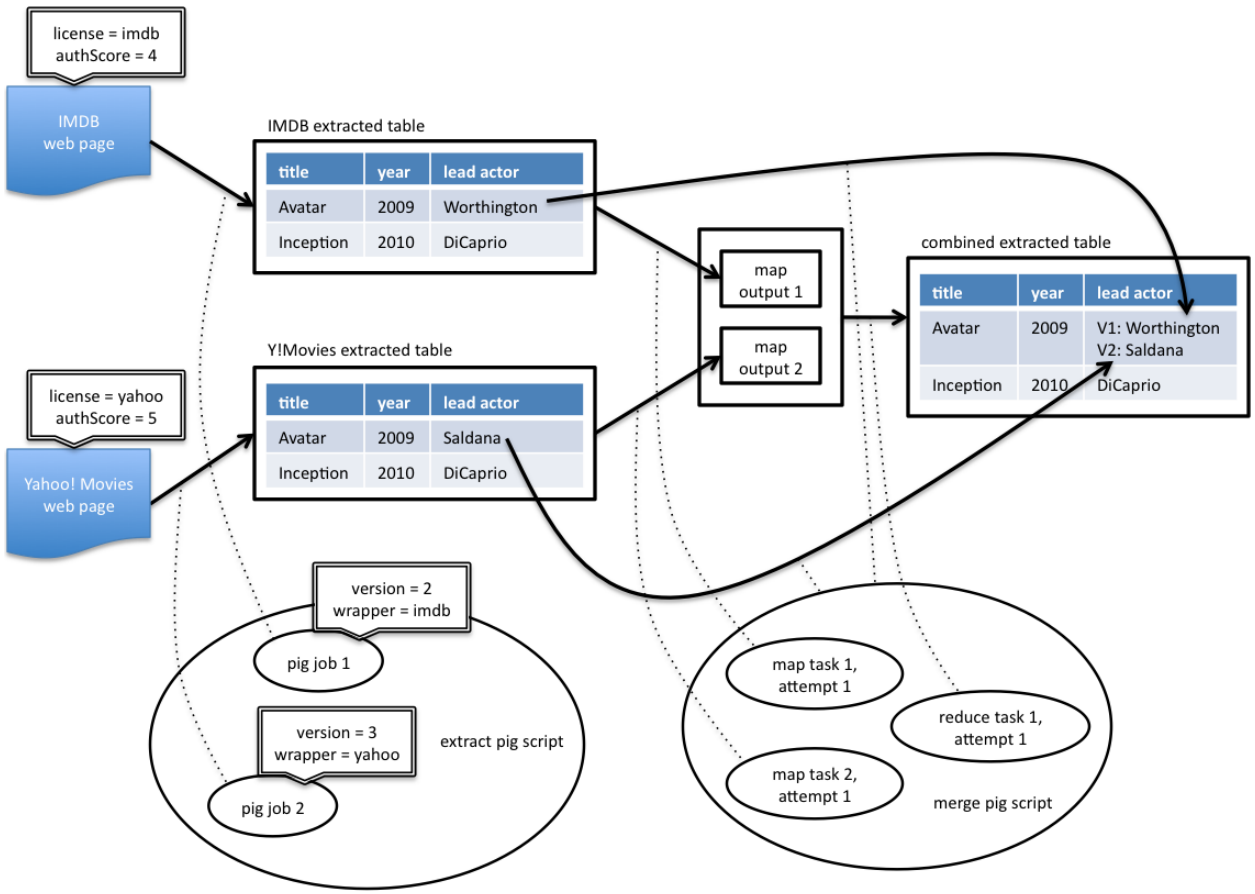


Figure 2: Example provenance graph.

has been extracted from two web pages (IMDB and Yahoo! Movies) and then merged. Inconsistencies have been preserved, and stored as alternate versions of cells in the merged table (the two web pages differed on the lead actor for the film "Avatar").

Formally, a graph vertex $V = (id, T, \{e_1, e_2, \ldots, e_k\}, \{v_1, v_2, \ldots, v_m\})$ is defined by a globally unique id, a type T, the ids of one or more complex element e_i of type T, and a value v_j for each attribute in T's attribute set. Each vertex represents the union of a set of complex data or process elements of a given type. A common case involves sets of size one ($k = 1$), e.g. (12, { row, column group }, { 8 }, { owner "Jeff", location "Singapore data center" }). Another example, with $k = 2$ (but no attribute values, also common), is (14, { MR task }, { 9, 10 }, { }) where 9 and 10 refer to complex elements (9, { map task }, { map task 1 }) and (10, { map task }, { map task 2 }), respectively. Figure 2 has one vertex with $k = 2$: the rectangle surrounding "map output 1" and "map output 2." Most vertices in Figure 2 have no attributes; exceptions are: web pages (license and authority score); extract pig jobs (version of extract script used, wrapper parameter).

Connections among graph vertices take the form of three-way (d_1, p, d_2) relationships, denoting that process element p produced data element d_2 by reading data element d_1. More particularly, *part of* p produced *all of* d_2 by reading *part of* d_1. (These semantics stem from the fact that creation of a data "touches" every byte of the data, whereas reading data and executing code rarely touch all the data/code (e.g. indexes and column stores; code branches).)[4]

In Figure 2, each provenance relationship (d_1, p, d_2) is shown as a dark arrow (d_1 to d_2 link) combined with a light dotted arrow (link to p). The provenance relationships on the left-hand side of Figure 2 are coarse-grained in terms of data links, and semi-coarse-grained in terms of process links (pig jobs that ran a particular version of the pig script called "extract," with a particular web page wrapper). The provenance relationships on the right-hand side occur at two granularities: (1) fine-grained links from data cells in the IMDB and Yahoo! Movies tables to cell versions in the combined extracted table, with coarse-grained references to the "merge" pig script; (2) coarse-grained links from the IMDB and Yahoo! Movies tables to the combined extracted table (via intermediate map output files), with fine-grained references to the specific map and reduce task attempts that handled the data.

4. OPEN-WORLD SEMANTICS

Recall from Section 1.1 that Ibis makes an *open-world* assumption about the metadata it manages. Here we formally

[4]We have found these semantics to suffice for the applications we have considered, but of course if needed one could always expose the control over the *part/all* semantics of each provenance connection to users.

define open-world semantics in the context of Ibis.

Let M denote the metadata currently registered with an Ibis instance. M encodes a set $\mathcal{F}$ of *facts*, such as the known set of data and process elements, their containment relationships, and the known provenance linkages. Ibis assumes that $\mathcal{F}$ is *correct* but (possibly) not *complete*, i.e. there exists some *true world* of facts $\hat{\mathcal{F}} \supseteq \mathcal{F}$. Let the *extension* $ext(\mathcal{F})$ denote the set of all facts that can be derived from $\mathcal{F}$ and are guaranteed to be part of any true world that is consistent with $\mathcal{F}$, i.e. $\mathcal{F} \subseteq ext(\mathcal{F}) \subseteq \hat{\mathcal{F}}$. ($ext(\mathcal{F})$ consists of all *certain* facts, analogous to *certain answers* in standard open-world semantics [9].)

Examples of facts in $ext(\mathcal{F})$ that are not in $\mathcal{F}$ include inferred containment relationships for complex elements, and transitively inferred provenance links. As an example of a fact that may be in $\hat{\mathcal{F}}$ but is not in $ext(\mathcal{F})$, suppose $\mathcal{F}$ includes "process p emitted row r_1," "process p emitted row r_2," and "r_1 and r_2 are part of table T"; even if $\mathcal{F}$ mentions no rows in T other than r_1 and r_2, the assertion "process p emitted the entire table T" cannot be included in $ext(\mathcal{F})$ because of the possibility that T contains additional rows in the true world $\hat{\mathcal{F}}$.

Ibis queries are answered with respect to $ext(\mathcal{F})$. In other words, the answer to query Q is equivalent to the one produced by the following two-step evaluation procedure: (1) derive and materialize $ext(\mathcal{F})$; (2) answer Q by performing "lookups" into $ext(\mathcal{F})$. These steps are the subjects of Sections 5 and 6, respectively.

Note that "positive queries" (which lookup facts that are implied by $ext(\mathcal{F})$) yield certain-answers semantics, but "negative queries" (which lookup facts that are not implied by $ext(\mathcal{F})$) such as ones that use "NOT EXISTS" or "MAX" do not, because some facts that cannot be derived based on Ibis's knowledge may be correct in the true state of the world. For completeness, our query language described in Section 6 does permit negative constructs. In practice they should either be disallowed, or come with a disclaimer about the deviation from certain-answers semantics. Another possibility is to record facts about completeness (i.e. the relationship between $ext(\mathcal{F})$ and $\hat{\mathcal{F}}$) such as "all table/job-level provenance links are being captured," and use them to vet negative queries; developing such an approach is left as future work.

5. INFERRING RELATIONSHIPS

Ibis's core strength is its ability to infer relationships among components of the provenance graph that span granularities. This section gives formal definitions of predicates that Ibis can infer with certainty (i.e. $ext(\mathcal{F})$, defined in Section 4). Let $\mathcal{V}$ denote the set of provenance graph vertices currently registered with an Ibis instance. Under open-world semantics (Section 4) we must assume the existence of some vertex set $\mathcal{V}' \supseteq \mathcal{V}$ (along with additional provenance relationships) that captures the real situation. Ibis's relationship inference semantics are defined in the context of $\mathcal{V}'$.

5.1 Under

Central to reasoning about granularity-spanning metadata is the *under* predicate, which determines whether the data or process element described by one vertex V_1 is contained in the element described by another vertex V_2. For example, in Figure 2 the cell containing `Worthington` is under the IMDB extracted table's `lead actor` column, which in turn is under the IMDB extracted table.

DEFINITION 5.1 (UNDER). *Given two provenance graph vertices V_1 and V_2 with complex element sets $E(V_1)$ and $E(V_2)$, V_1 is* under *V_2 iff $\nexists \mathcal{V}' \supseteq \mathcal{V}$ such that $\bigcup_{e_1 \in E(V_1)} \mathcal{G}(e_1) \nsubseteq \bigcup_{e_2 \in E(V_2)} \mathcal{G}(e_2)$*[5].

Fortunately, there exists an efficient way of checking whether a pair of vertices satisfies the under predicate using just the known vertex set $\mathcal{V}$, which is equivalent to the above definition (a proof of equivalence is given in Appendix A):

DEFINITION 5.2 (EFFICIENT UNDER CHECK). *Given two provenance graph vertices V_1 and V_2 with complex element sets $E(V_1)$ and $E(V_2)$, V_1 is under V_2 iff $\forall e_1 \in E(V_1), \exists e_2 \in E(V_2)$ such that e_1 is under e_2, with the under predicate defined over complex elements as follows: Given two complex elements e_1 and e_2 with basic element sets $B(e_1)$ and $B(e_2)$, e_1 is under e_2 iff $\forall b_2 \in B(e_2), \exists b_1 \in B(e_1)$ such that b_2 contains*[6] *b_1.*

5.2 Feeds, Emits and Influences

Recall the three-way provenance relationships introduced in Section 3.3: a relationship (d_1, p, d_2) denotes that (part of) processing element p produced (all of) data element d_2 by reading (part of) data element d_1. Ibis can answer three types of predicates over the set of registered provenance relationships:

- **Data *feeding* a process:** Given data element d and process element p, does (part of) d *feed* (part of) p?
- **A process *emitting* data:** Given data element d and process element p, does (part of) p *emit* (all of) d?
- **Data *influencing* other data:** Given two data elements d_1 and d_2, does (part of) d_1 *influence* (all of) d_2, either directly (*influences(1)*) or indirectly (*influences(k)*)?

Formal definitions and examples follow:

DEFINITION 5.3 (FEEDS). *Data vertex d* feeds *process vertex p iff there exists a provenance relationship (d', p', d_2) such that d' is under d and p' is under p.*

In our example provenance graph shown in Figure 2, from the relationship (`IMDB web page`, `pig job 1`, `IMDB extracted table`) we can infer that (part of) `IMDB web page` feeds (part of) `extract pig script`. From the relationship (`Worthington`, `merge pig script`, `V1: Worthington`) we can infer that (part of) `row(Avatar, 2009, Worthington)` feeds (part of) `merge pig script`.

DEFINITION 5.4 (EMITS). *Process vertex p* emits *data vertex d iff there exists a provenance relationship (d_1, p', d') such that p' is under p and d is under d'.*

Again considering Figure 2, from the relationship (`IMDB web page`, `pig job 1`, `IMDB extracted table`) we can infer that (part of) `extract pig script` emits (all of) `IMDB extracted table`, and also that (part of) `pig job 1` emits (all of) `row(Avatar, 2009, Worthington)`.

DEFINITION 5.5 (INFLUENCES). *Given two data vertices d_1 and d_2: d_1* influences(0) *d_2 iff d_2 is under d_1;*

[5]Recall the definition of granularization ($\mathcal{G}(\cdot)$) from Section 3.2.

[6]Recall the containment definition from Section 3.2.

d_1 influences(1) d_2 *iff* d_1 *influences(0)* d_2 *or there exists a provenance relationship* (d_1', p, d_2') *such that* d_1 *influences(0)* d_1' *and* d_2' *influences(0)* d_2*; for any integer* $k > 1$, d_1 influences(k) d_2 *iff there exists a vertex* d^* *such that* d_1 *influences(1)* d^* *and* d^* *influences*$(k-1)$ d_2.

The influence relationships in Figure 2 include:

- (part of) `IMDB extracted table` influences(0) (all of) `row(Avatar, 2009, Worthington)`
- (part of) `row(Avatar, 2009, Worthington)` influences(1) (all of) `V1: Worthington`
- (part of) `IMDB web page` influences(1) (all of) `lead actor column of IMDB extracted table`
- (part of) `IMDB web page` influences(2) (all of) `V1: Worthington`

An example of an inference that cannot be made is: (part of) `IMDB extracted table` influences(k) (all of) `combined extracted table` (for any value of k).

6. QUERY LANGUAGE

We now turn to Ibis's query language, called *IQL*. Given the concepts introduced above and knowledge of SQL, the query language itself is fairly straightforward. Therefore, in lieu of a tedious exhaustive description of IQL, we give an overview of the main language constructs and illustrate their use via a number of examples.

IQL starts with SQL and makes the following modifications:

- The FROM clause references complex element types, e.g. Row or (Row, Column). IQL also supports special wildcards: AnyData (any data type), AnyProcess (any process type) and Any (any data or process type).
- The SELECT and WHERE clauses can reference a special `id` field, as well as elements of each type's attribute set (for wildcards, no attributes are accessible).
- The *union*, *feeds*, *emits* and *influences(k)*[7] predicates (defined in Section 5) can be used in the WHERE clause.

Table 1 gives some example query/answer pairs formulated over the example provenance graph in Figure 2. The first four queries in the table are inspired by data workflow debugging scenarios encountered at Yahoo. The fifth query corresponds to a Yahoo use-case involving content licensing: each data source comes with a license that restricts the contexts in which data derived from it can be displayed to end-users, and provenance is used to perform last-mile filtering for a given context. The final query is a somewhat contrived variation of the license example, which instead filters by source authority score; it shows a more elaborate use of our language.

Logically speaking, IQL queries are evaluated over the extended database $ext(\mathcal{F})$, defined in Section 4. As with SQL, IQL query semantics are equivalent to the following three-step evaluation strategy (in IQL's case, over $ext(\mathcal{F})$): (i) evaluate the the FROM clause to construct the cross-product of the sets of elements referenced; (ii) apply the filters given in the WHERE clause; and (iii) apply the projections specified in the SELECT clause.

[7]We do not currently support unbounded transitive closure ($k = \infty$), but support for this feature could be added via recursive query processing techniques.

7. PROTOTYPE IMPLEMENTATION

We have built a simple implementation of a storage and query manager for Ibis, on top of a conventional relational database system (SQLite [12]) using query rewriting from IQL into SQL. The purpose of this implementation is to test the applicability and ease-of-use of our model and query language, not to serve as an efficient or scalable system for managing provenance metadata—that is future work.

7.1 Relational Encoding

Our prototype uses a very simple encoding of Ibis's information into relational tables:

- **Gsets:** The gsets are stored using two tables: `gnodes(type, granularity)` stores granularity nodes in the attribute `granularity`, with `type` being 'Data' or 'Process'; each edge depicting a containment relationship between nodes of gsets are stored in table `gcont(child, parent)` in the obvious way.
- **Simple Elements:** The `simpleElements(seId, granularity)` table stores for every simple element identified by `seId`, the granularity in the gset given by `granularity`.
- **Complex Elements:** Table `complexElements(ceId, seId)` stores the mapping from complex elements to simple elements: a complex element `ceId` comprised of n simple elements is represented as n tuples `(ceId, seId`$_i$`)`.
- **Provenance Graph:** Vertices of the provenance graph (corresponding to sets of complex elements) are stored in table `vertices(vertexId, ceId)`, which associates a set of complex ids with each vertex, analogous to the way `complexElements` associates a set of simple ids with each complex id.

  Three-way provenance relationships are represented in the table `edges(src_data, process, dst_data)`, where `src_data` and `dst_data` are the source and destination data vertex identifiers and `process` gives the process vertex identifier.
- **Attributes:** Every complex element type `X` has a separate table `X_attrs(nodeid, <attributes>)` to store attributes. For example, in our demo scenario we have a table `Webpage_attrs(vertexId, license, authScore)` that maintains the attributes `license` and `authScore` for every webpage identified by `vertexId`.
- **Under:** Under relationships for simple elements are maintained using the `under(src, dst)` table in the obvious way.

7.2 Query rewriting

The four Ibis-specific constructs used to augment SQL—under, influences, feeds, emits—are automatically converted to SQL as follows. The under construct for vertices in the provenance graph (representing sets of complex elements) is converted to a SQL query over the `under` table (over simple elements) using the check from Definition 5.2. Feeds and emits are easily converted based on a lookup of the provenance tables. Note, however, that the translation of feeds and emits also needs to add the `under` table: For example, recall from Definition 5.3 that data vertex d *feeds* process vertex p when there is a provenance relationship (d', p', d_2) such that d' is under d and p' is under p. Finally, influences(k) is translated to a chain of joins using `under` and `edges`.

description	query	answer(s)
Find web pages that influence V1 of the Avatar lead actor field in the combined extracted table.	**select** p.id **from** WebPage p, Version v **where** p **influences(2)** v **and** v.id = (Avatar lead actor V1);	IMDB web page
Find data items that influence the combined extracted table.	**select** d.id **from** AnyData d, Table t **where** d **influences(2)** t **and** t.id = (combined extracted table);	{ map output file 1, map output file 2 } combined extracted table
Suppose the first attempt of the second map task of the merge job experienced a problem that is suspected to stem from malformed input data. Find the data table it read.	**select** t.id **from** MRTaskAttempt a, Table t **where** t **feeds** a **and** a.id = (merge pig script map task 2 attempt 1);	Y!Movies extracted table
Suppose version 3 of the extract pig script is found to have a bug. Find all "contaminated" data tables, i.e. ones containing data that stems from that version of the script.	**select** t.id **from** PigScript p, PigJob j, AnyData d1, AnyData d2, Table t **where** p.id = (extract pig script) **and** j **under** p **and** j.version = 3 **and** j **emits** d1 **and** d1 **influences(2)** d2 **and** d2 **under** t;	Y!Movies extracted table combined extracted table
Filter versions of combined table cells—only retain ones derived solely from sources with the Yahoo license.	**select** v.id **from** Version v, Table t **where** t.id = (combined extracted table) **and** v **under** t **and** **not exists** (**select** * **from** WebPage source **where** source **influences(2)** v **and** source.license != 'yahoo');	Avatar lead actor V2
Resolve cell version ambiguity by selecting the one that derives from the most authoritative web page.	**select** v.id, source.authScore **from** Version v, WebPage source **where** source **influences(2)** v **and** source.authScore = (**select** max(source2.authScore) **from** Version v2, WebPage source2, (Row,Column) commonParent **where** source2 **influences(2)** v2 **and** v **under** commonParent **and** v2 **under** commonParent);	id = Avatar lead actor V2; authScore = 5

Table 1: Example IQL queries, and the answers with respect to Figure 2.

8. CONCLUSION

Motivated by data processing systems comprised of multiple independently-developed sub-systems, we have developed a metadata and data provenance management service called Ibis. Ibis handles provenance metadata that spans multiple granularities of data and processing elements, and Ibis's query language is able to infer provenance relationships across granularities. Ibis is fully implemented using query rewriting on top of a conventional RDBMS. Future work will focus on efficiency and scalability issues—in regard to storing and querying provenance metadata, as well as capturing and shipping it from various systems.

Acknowledgments

We would like to thank the various members of the Yahoo content platform teams who shared information about debugging, license management and data provenance. We also thank Philip Bohannon, Andreas Neumann and Kenneth Yocum for their input on this project.

9. REFERENCES

[1] Apache. Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[2] Apache. Oozie: Hadoop workflow system. http://issues.apache.org/jira/browse/HADOOP-5303.

[3] Apache. Pig: High-level dataflow system for hadoop. http://hadoop.apache.org/pig.

[4] S. Bowers, T. M. McPhillips, and B. Ludascher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[5] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[6] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using ZOOM. *Concurrency and Computation: Practice and Experience*, 20(5):497–506, 2008.

[7] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludascher, T. M. McPhillips, S. Bowers, M. K. Anand, and

J. Freire. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007.

[8] M. Y. Eltabakh, W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva. Supporting annotations on relations. In *Proc. EDBT*, 2009.

[9] A. Y. Levy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[10] Moreau (Editor), L., Plale, B., Miles, S., Goble, C., Missier, P., Barga, R., Simmhan, Y., Futrelle, J., McGrath, R., Myers, J., Paulson, P., Bowers, S., Ludaescher, B., Kwasnikowska, N., Van den Bussche, J., Ellkvist, T., Freire, J. and Groth, P. The Open Provenance Model (v1.01). *Technical Report*, 2008.

[11] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In *Proc. USENIX Annual Technical Conference*, 2009.

[12] Open-Source Community. SQLite: An open-source database management library. `http://www.sqlite.org/`.

[13] D. Srivastava and Y. Velegrakis. Intensional associations between data and metadata. In *Proc. ACM SIGMOD*, 2007.

[14] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *Proc. Intl. Conference on Ontologies, Databases and Applications of Semantics*, 2003.

APPENDIX

A. PROOF OF EQUIVALENCE OF UNDER DEFINITIONS

Here we prove the equivalence of Definition 5.1 and Definition 5.2. We start by proving the equivalence for the case when $E(V_1)$ and $E(V_2)$ each consists of a single complex element (Section A.1), then extend the proof to sets of complex elements (Section A.2).

A.1 Under for single complex elements

Let $E(V_1)$ and $E(V_2)$ contain single complex elements e_1 and e_2 respectively. From Definition 3.5, we know that there exists some $\mathcal{V}' \supseteq \mathcal{V}$ such that $\mathcal{G}(e_1) \neq \emptyset$. So we prove the result when $\mathcal{G}(e_1) \neq \emptyset$. From Definition 5.2, for the basic element sets $B(e_1)$ and $B(e_2)$, e_1 is under e_2 iff the following condition is satisfied: $\forall b_2 \in B(e_2), \exists b_1 \in B(e_1)$ such that b_2 contains b_1. We prove the sufficiency and necessity of checking this condition:

Sufficient: For each $b_j \in B(e_2)$, let $b_{i(j)} \in B(e_1)$ satisfy the condition: That is, b_j is contained in $b_{i(j)}$. Therefore, we have

$$\mathcal{G}(b_{i(j)}) \subseteq \mathcal{G}(b_j)$$

Therefore, we have:

$$\mathcal{G}(e_1) = \bigcap_{b_i \in B(e_1)} \mathcal{G}(b_i) \subseteq \bigcap_{b_{i(j)} \in B(e_1)} \mathcal{G}(b_{i(j)}) \subseteq \bigcap_{b_j \in B(e_2)} \mathcal{G}(b_j) = \mathcal{G}(e_2)$$

Necessary: We prove necessity by contradiction. Suppose $b_j \in B(e_2)$ such that $\forall b_i \in B(e_1)$ we have that b_i is not contained in b_j, i.e., $\mathcal{G}(b_i) \not\subseteq \mathcal{G}(b_j)$. We have two cases: (1) Suppose b_j is a basic element of the finest granularity, then e_1 cannot be under e_2 since $\mathcal{G}(e_1) \neq \emptyset$, $b_j \notin \mathcal{G}(e_1)$, and $\mathcal{G}(e_2) \subseteq \{b_j\}$. (2) Suppose b_j is a basic element of granularity coarser than g_{min}. Then consider the completion $\mathcal{V}'$ of data obtained by adding distinct basic elements of granularity g_{min} under all basic elements of coarser granularity. We shall then then have $\mathcal{G}(e_1) \not\subseteq \mathcal{G}(e_2)$, since $\mathcal{G}(b_j)$ will not contain $\mathcal{G}(e_1)$.

A.2 Under for sets of complex elements

Next we show that the equivalence of Definitions 5.1 and Definition 5.2 for sets of complex elements $E(V_1)$ and $E(V_2)$ easily follows from the proof of the single-element case. Below we show equivalence by proving the "if" and "only if" portions of the condition in Definition 5.2 separately.

If: Clearly if $\forall e_1 \in E(V_1), \exists e_2 \in E(V_2)$ such that e_1 is under e_2 (based on the condition for single elements), we have $\bigcup_{e_1 \in E(V_1)} \mathcal{G}(e_1) \subseteq \bigcup_{e_2 \in E(V_2)} \mathcal{G}(e_2)$.

Only If: We prove by contradiction. Suppose $\exists e_1 \in E(V_1)$ such that $\forall e_2 \in E(V_2)$ we have e_1 is not under e_2, i.e., $\mathcal{G}(e_1) \not\subseteq \mathcal{G}(e_2)$. We can therefore construct $\mathcal{V}'$ as follows. We add a simple element X of the finest granularity under e_1 such that $\forall e_2 \in E(V_2)$, we have $X \notin \mathcal{G}(e_2)$. Therefore, $X \in \bigcup_{e_1 \in E(V_1)} \mathcal{G}(e_1)$ but $X \notin \bigcup_{e_2 \in E(V_2)} \mathcal{G}(e_2)$. Therefore, $\bigcup_{e_1 \in E(V_1)} \mathcal{G}(e_1) \not\subseteq \bigcup_{e_2 \in E(V_2)} \mathcal{G}(e_2)$.

Answering Queries using Humans, Algorithms and Databases

Aditya Parameswaran
Stanford University
adityagp@cs.stanford.edu

Neoklis Polyzotis
Univ. of California at Santa Cruz
alkis@cs.ucsc.edu

ABSTRACT

For some problems, human assistance is needed in addition to automated (algorithmic) computation. In sharp contrast to existing data management approaches, where human input is either ad-hoc or is never used, we describe the design of the first declarative language involving human-computable functions, standard relational operators, as well as algorithmic computation. We consider the challenges involved in optimizing queries posed in this language, in particular, the tradeoffs between uncertainty, cost and performance, as well as combination of human and algorithmic evidence. We believe that the vision laid out in this paper can act as a roadmap for a new area of data management research where human computation is routinely used in data analytics.

Keywords

Human Computation, Crowdsourcing, Declarative Queries, Query Optimization, Uncertain Databases

1. INTRODUCTION

Large-scale human computation (or, crowd-sourcing) services, such as Mechanical Turk [3], oDesk [5] and SamaSource [6], harness human intelligence in order to solve tasks that are relatively simple for humans–identifying and recognizing concepts in images, language or speech, ranking, summarization and labeling [10], to name a few–but are notoriously difficult for algorithms. By carefully orchestrating the evaluation of several simple tasks, enterprises have also started to realize the power of human computation for more complex types of analysis. For instance, Freebase.com has collected over 2 million human responses for tasks related to data mining, cleansing and curation [24]. Several startups, such as CrowdFlower [1], uTest [7] and Microtask [4], have developed a business model on task management for enterprises. In addition, the research community has developed libraries that provide primitives to create and manage human computation tasks and thus enable programmable access to crowd sourcing services [25, 20, 26].

These recent programmatic methods are clearly important and useful, but we believe that they cannot scale to the increasing complexity of enterprise applications. Essentially, these procedural methods suffer from the same problems that plagued data management applications before declarative query languages. The onus falls on the programmer/developer to manage which tasks are created and evaluated, deal with discrepancies in the received answers, combine these answers with existing databases, and so on. Instead, we propose a declarative approach: the programmer specifies what must be accomplished through human computation, and the system transparently optimizes and manages the details of the evaluation.

Complex Task Example. We will use an example inspired by the popular restaurant aggregator service Yelp in order to illustrate the spirit of our approach. Assume that Yelp wishes to display a few images for each restaurant entry, picked from a big database of user-supplied images. The selected images must not be dark, must not be copyrighted, and they must either display food served in the restaurant or display the exterior of the restaurant with its name in view.

Yelp's developer can write a procedural application that accesses each image in the database, checks on its copyright restrictions (perhaps using another stored database), and then applies the remaining predicates using a crowd-sourcing service like Mechanical Turk. The application may use a toolkit like Turkit [25] in order to interface with Mechanical Turk. In addition to using the Mechanical Turk, the application may use computer vision algorithms in order to determine whether an image is dark or not, to augment human computation with algorithmic computation of lower latency. Of course, the algorithms may not be able to handle accurately all images, and hence the developer must make a judicious choice between the two options for different images. He/she must also orchestrate the composition of these simple tasks and determine the order in which they are evaluated and on which images. The developer also has to ensure that Mechanical Turk tasks are priced appropriately and that the total budget remains within limits. Furthermore, he/she has to deal with discrepancies in the answers returned by human workers–not all people may agree on what is a dark image–and to determine whether the uncertainty in the answers is acceptable.

Overall, this simple example illustrates the large complexity in developing even simple applications. It is straightforward to see that this programming model is unsustainable for complex tasks that need to combine information from databases, human computation and algorithms.

The Declarative Approach. Our proposed approach views the crowd sourcing service (CrSS for short) as another database where facts are computed by human processors. By promoting the CrSS to a first-class citizen on the same level as extensional data, it is possible to write a declarative query that seamlessly combines information from both. In our example, the developer has to only specify that the in-database images will be filtered with predicates

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9–12, 2011, Asilomar, California, USA.

that are computed using the Mechanical Turk or algorithms. The system becomes responsible for optimizing the order in which images are processed, the order in which tasks are scheduled, whether tasks are handled by algorithms or a CrSS, the pricing of the latter tasks, and the seamless transfer of information between the database system and the external services. Moreover, it provides built-in mechanisms to handle uncertainty, so that the developer can explicitly control the quality of the query results. Using the declarative approach, we can facilitate the development of complex applications that combine knowledge from human computation, algorithmic computation, and data.

Existing database technology can serve as the starting point to realize this novel paradigm, but we believe that new research is required in order to fully support it. Specifically, the declarative access of crowd sourcing services has a unique combination of features: the query processor must optimize both for performance and monetary cost (tasks posted to the CrSS cost money); the obtained data is inherently uncertain, since humans make mistakes or have different interpretations of the same task; the resulting uncertainty cannot be handled by existing techniques; and, the latency of human- and algorithmic-based computation is significant and also unpredictable. This combination is sufficiently complex to require a redesign of the query processor, creating fertile ground for new and innovative research in data management.

The goal of this paper is to identify this research opportunity, perform an initial demarkation of the problem space, and to propose some concrete research directions.

2. CROWD-SOURCING SERVICES: BASICS

A CrSS allows users to post unit *tasks* for humans to solve. A task comprises two main components, namely, a description and a method for providing an *answer*. For instance, consider the task of examining an image in order to determine whether it depicts a clean location. The description may be formed by the image along with the text "Is this location clean?", and the method for providing an answer can be simply a Yes/No radio button. The CrSS is responsible for posting the task and returning the answers of the human workers back to the application.

An examination of popular crowd-sourcing services reveals a wide variety of posted tasks. The degree of difficulty varies widely as well, from simple tasks that require a few minutes of work, to elaborate tasks that may take several hours to complete. Such complex tasks typically involve content creation, e.g., write an article on a topic or review a website. In this paper, we focus on simple tasks for two reasons: they can form the building blocks of more complex tasks, and due to their short duration they are more likely to be picked up by human workers. Extending our ideas to complex creative tasks is an interesting direction for future work.

There are three more components to a unit task: (a) the monetary reward for providing the answer, (b) the number of workers who must solve the task independently before it is deemed completed (in order to reduce the possibility of mistakes), and (c) any criteria that a worker must fulfill in order to attempt the task. Several previous studies have examined the effect of these components on the quality of the obtained answers, and have developed guidelines for their configuration [28, 21, 22]. In this paper, we adopt these guidelines and do not consider these components further.

It is important to note that, depending on the particular CrSS, there may be significant latency in obtaining answers to posted tasks. However, we can expect this latency to be reduced in subsequent years, as crowd-sourcing services gather a larger population of human workers and provide better interfaces for matching tasks to interested workers. In the same direction, several startup companies are aiming to create a more efficient marketplace for matching requesters to workers. (It is interesting to note that there has been a recent surge in crowd-sourcing startups, with more than 20 companies established in the past 5 years.)

3. DECLARATIVE QUERIES OVER HUMANS, ALGORITHMS AND DATA

The crux of our proposal is to employ a declarative query language in order to describe complex tasks over human processors, data and algorithms. In this section, we describe more concretely the features of this language and lay the groundwork for defining its semantics and evaluation methods in subsequent sections.

Query Model. We illustrate the query model using a Datalog-like formalism. We stress that this is done for notational convenience and not because we rely on Datalog's features. (It is unlikely that we will encounter recursive tasks in practice.)

Let us consider the following example complex task: *Find photos that are large, in jpeg format, and depict a clean beach or a city that is neither dangerous nor dirty.* The task can be encoded as the following declarative query in our formalism:

```
travel(I):= rJpeg(I), hClean(I), hBeach(I), aLarge(I)
travel(I):= rJpeg(I), hClean(I), haCity(I,Y), rSafe(Y)
```

Let us explain each construct in more detail, starting with the first rule and examining the predicates left-to-right.

- We assume an extensional database table, named `rJpeg`, that contains jpeg images. Hence, `rJpeg(I)` becomes true for every binding of `I` to an input image. (We overload `I` to represent both the image ID as well as the image itself, in order to simplify notation. The use should be clear from the context.)
- The predicate `hClean` is termed an h-predicate, as it is evaluated using a CrSS. The idea is that, for a given binding to variable `I`, `hClean(I)` is evaluated by posting a task to the CrSS–"Does this image depict a clean location?"–and retrieving the answer. We discuss the instantiation of such tasks later.
- Similarly, there is an h-predicate `hBeach(I)` for checking whether the image depicts a beach, which is evaluated with separate tasks.
- Finally, the predicate `aLarge` is termed an a-predicate and is evaluated by invoking an algorithm with the binding for `I` as an input argument. For instance, the algorithm may examine the image file to compute the number of pixels.

The second rule employs the same first two predicates and an additional extensional predicate `rSafe` which is a table of known safe cities. An interesting feature here is `haCity` which is a *hybrid* ha-predicate–it can be evaluated by a human worker or by invoking a specific algorithm. In the case of a human worker, the task will show the image bound to variable `I` and will ask for the name of the depicted city. This value will be returned in variable `Y`. An algorithmic evaluation will take `I` as input and generate `Y` as output.

We assume that the application developer provides templates in order to map a h-predicate to unit tasks, as described in §2. While unit task design is an extremely important issue, we expect to leverage work from the HCI and Questionnaire Design communities to design effective user interfaces corresponding for each unit task. The choice of the unit tasks posed for a given h-predicate can have an effect on both cost and latency, as we discuss in §6.4. To simplify presentation, we assume a fixed mapping from h-predicates to unit tasks for the remainder of the paper.

Consider the example of `haCity(I, Y)`. In this case, the template may have an image placeholder populated with the binding for `I`, a message "Which city (if any) is depicted in this image?", and a text box to write the name of the city. The value of the text box becomes the binding for `Y`. Similar templates are provided for a-predicates. Clearly, the provided templates also provide bound/free annotations for the variables in Datalog predicates. Returning to the example of `haCity(I,Y)`, it is meaningful to bind `X` and let the humans compute `Y`, while the other way around does not make much sense. These annotations clearly constrain the order in which predicates can be evaluated.

Semantics for hQuery. How do we define the results of an hQuery? This would be straightforward if we were dealing with a regular Datalog query: we can define what constitutes a correct output tuple, and then define the semantics of the query as the complete set of correct output tuples. However, neither of these definitions are obvious in our context.

The definition of a *correct* output tuple is not straightforward, since discrepancies arise naturally in the context of h- and a-predicates. Let us consider again the example of travel images. Different human workers may have different notions of cleanliness, hence giving different answers for the same `hClean(I)` predicate. It may also be possible to make mistakes, e.g., answer `hCity(I,Y)` with the wrong city for a given image. To complicate matters further, discrepancies may be correlated, e.g., if a human worker answers negatively for `hClean(I)` when `I` is bound to a beach image, then he/she may be more likely to answer positively when `I` is bound to a city image. Overall, it may be possible to both assert and negate `travel(I)` depending on which human workers or algorithms are used to evaluate the predicates.

Clearly, it is necessary to define a "consolidation" mechanism for discrepancies in order to define correctness. A simple mechanism might be majority voting – have several human workers (or algorithms) tackle the same question and then take the most frequent answer. We may even employ a more elaborate scheme where (potentially conflicting) answers are assigned probabilities, and correct answers are defined based on a probability threshold. In the coming sections, we consider several possibilities of varying complexity and scope. For the time being, we assume that some mechanism is employed to define what constitutes a correct output tuple.

The second complication is that it may not be desirable to obtain all output tuples. In some cases, the application may impose a budget on the total cost of CrSS tasks, which limits the number of h-predicates that can be evaluated and hence the number of output tuples. In other cases, it may be sufficient to obtain any k output tuples, for some fixed k, in order to avoid the high latency and monetary cost of computing all output tuples. We thus arrive at three different ways that the application can obtain the output of an hQuery:

1. Return all correct output tuples while minimizing the number of h-predicate evaluations and total time
2. Return any k correct output tuples while minimizing the number of h-predicate evaluations and total time, for fixed k.
3. Maximize the number of correct output tuples and minimize total time for a maximum of m h-predicate evaluations, for fixed m.

These definitions makes the simplifying assumption that all tasks are priced equally. This is a reasonable compromise, since applications are likely to use well established guidelines for pricing unit tasks [21, 28]. However, the problem of dynamic pricing is still interesting and we consider as a possible extension of our framework.

Other Examples. We now present three other examples to illustrate various aspects of our declarative query model.

Example 1: Consider the query: *Find all images of people at the scene of the crime who have a known criminal record.* This query can be written as:

```
names(A, I):= rCriminal(Y, A), rCand(I), haSim(I,Y)
```

The EDB table `rCriminal` contains images of known criminals as well as their names. The candidate table `rCand` contains images of people witnessed at the scene of a crime. The hybrid predicate `haSim` tries to evaluate if the person in the candidate table is also present in the criminal table by performing a fuzzy image match. This corresponds to the unit task " Do these two images depict the same person?". Alternatively, a face recognition algorithm may be used to determine whether the two images are of the same person.

Note also that our unit tasks need not have a one-one relationship with the human predicates. We consider this dimension of query optimization in more detail in §6.4. For instance, we may post k images from the candidate table in one column as well as k images from the criminal table in the second column and pose the task description "Match images from the first column to images from the second column that are of the same person." In this case, the relationship is one-many.

The results of a similarity join query such as the one given above can be used to derive a result for entity resolution, clustering or projection (with duplicate elimination).

Example 2: Subsequently, in results of the query above, we may wish to find the best image corresponding to each potential criminal `A`. This query can be posed as follows:

```
topImg(A, hBest(<I>)):= names(A, I)
```

In this query, we use an aggregation (i.e., group by) function on the `A`, and across all images for a given `A`, output the one that is the best (as evaluated by humans). Thus, the aggregation function applied on all the images for a given `A` is the h-predicate `hBest`. Note that in this case, we may need to post several tasks to ascertain which image is the best for a given name `A`. Thus the relationship is many-one between the posted tasks and the h-predicate.

Example 3: Sorting is another primitive that we would like to use in our queries. For instance, we may wish to sort the results corresponding to each `A` in the previous query instead of returning a single best image. Sorting can be expressed using additional syntactic machinery, as is done in prior work in datalog. The results of the sort are stored in a successor intensional database `succ(I, I')`, i.e., `I` appears immediately before `I'` in a sorted order. We can apply a selection condition to the results of a sorting operation to obtain the top-k results, for a given k.

Sorting and top-k are examples of second-order predicates, discussed in more detail in §6.3.

4. FIRST STEP: CERTAIN ANSWERS

Having defined the query language at a logical level, we consider next the design of a query processor to evaluate such queries.

Our starting assumption is that human processors and algorithms always answer questions precisely, i.e., no mistakes or discrepancies are observed. In other words, there is no need to ask a given question to more than one human or algorithm. The adopted assumption simplifies the semantics of the query language, as a query answer is any tuple that is derived through the query's rules based on the answers of humans and algorithms and the extensional database.

The certain-answers assumption is clearly unrealistic in several scenarios, but, as we discuss later, even this simplified case raises

interesting technical challenges for the design of a query processor. We lift this assumption in §5, when we discuss the evaluation of queries under various models of uncertainty.

Design Space. As we argued in §1, a conventional optimize-then-execute design is not appropriate for our problem, since it is very difficult to model the cost and distribution of answers from human processors and algorithms. We are thus led naturally to an adaptive query processing scheme, where any optimization is based on run-time observations.

What should the engine optimize for? Performance is clearly important, since in all cases we wish to minimize total running time. At the same time, it is equally important to optimize carefully the questions issued to the CrSS, since we also wish to minimize the total number of questions or we may even have a bound on the total number of questions. Thirdly, we may also want to maximize the number of correct output tuples when the number of questions is fixed. Essentially, we have a two-criteria optimization problem that involves two out of time, monetary cost and number of correct output tuples. This is a unique property of our problem statement that separates it from previous works in adaptive query processing.

Consider the case when we wish to retrieve all correct tuples. Selecting which questions to ask is a non-trivial problem and is in itself a very interesting direction for future work in this area. We revisit this point at the end of section where we outline some initial research directions. Based on our own work in this area [30], we can point out an interesting trade-off: we may employ a computationally expensive strategy, which adds to the total completion time of the query, in order to carefully select where to ask the fewest possible questions, or we may choose a cheaper strategy at the expense of asking more questions than necessary. It is very difficult to collapse the two dimensions in a single measure, and hence two different question-selection strategies may simply be incomparable. However, note that interesting strategies will lie along a skyline of low computational overhead and low number of questions.

Query Processing Engine. Based on the previous principles, we envision a query processing engine that operates in successive stages. In each stage, the engine performs computation solely on predicates of a specific type, i.e., intensional predicates, h-predicates, or a-predicates. For instance, the engine may first compute some intermediate results by using relational processing solely on extensional data, then issue some questions to the CrSS, then compute more intermediate results using relational operators, then evaluate a-predicates, continue with CrSS questions, and so on. (Since obtaining results for h-predicates may involve high latency, they will be evaluated asynchronously.) Hence, each stage generates more results by building on the available results of previous stages.

The stage-based approach separates the evaluation of predicates of different types and hence isolates the optimization decisions at two points: inter- and intra-stage. Inter-stage optimization determines which type of stage to run next and with what resources. For instance, consider a query that aims to maximize the number of answers under a limited number of questions. An intuitive optimization strategy may favor stages of relational and algorithmic processing, in order to generate query answers for "free", and will invoke CrSS stages infrequently and with a small number of questions each time. The choice of relational vs algorithmic stages may be driven by run-time statistics on the rate of generated results, since the aim is to maximize the number of query answers.

Intra-stage optimization depends on the type of the stage. For a purely relational stage, we can reuse any of the existing adaptive query processing techniques that optimize for performance. For a CrSS or an algorithmic stage, we need to develop strategies to select what questions to ask, as described earlier. These strategies can optimizer eithe for performance, in the case of algorithmic stages, or for a combination of performance and number of issued questions, in the case of CrSS stages.

What questions should be asked? Intuitively, we wish to ask questions whose answers provide the most "information gain", but the latter is non-trivial to quantify. However, we can define some useful heuristics that approximate this notion of gain.

- It is useful to prioritize questions based on the number of affected rules. For instance, we may prefer to ask about `hClean(`α`)` compared to `hClean(`β`)` if the former will enable the processing of more query rules.
- If we are interested in producing all the query answers, then it is prudent to ask questions that will reduce the volume of data to be processed in subsequent stages. Hence, selective questions are preferred.
- If, instead, for a fixed budget, we are interested in maximizing the number of query answers, then we want to ask questions whose answers are likely to lead to more results being generated in subsequent stages. Hence, non-selective questions are preferred.

The last two heuristics require some estimation of selectivity, which can be done in an obvious way based on observations on past questions. An interesting twist is to also consider data similarity in order to obtain more accurate estimates. For instance, consider again the choice between `hClean(`α`)` and `hClean(`β`)`, and assume that we are interested in non-selective questions (i.e., maximizing the number of query answers). Given a suitable data similarity function among image objects, we can examine which of α or β bears a higher resemblance to images for which past `hClean` questions have returned positive answers.

Overall, selecting the right questions is a non-trivial optimization problem in the development of an effective query processor. In our previous work, we developed several such strategies for specific classes of human-computable predicates that occur commonly in larger tasks [30]. Generalizing these strategies, and enhancing them by taking into account data similarity and the answers to previous questions, is part of our ongoing work in this area.

5. THE REALITY: UNCERTAINTY

So far, we have assumed that correct answers are provided by humans to unit tasks posted on the CrSS. However, this is hardly the case in practice, and we need to deal with incorrect and biased answers in most cases. In this section, we consider ways and means of dealing with and resolving this uncertainty.

We describe the sources of uncertainty in §5.1 and how to modify the ways to use hQuery in §3 to incorporate uncertainty. We describe three candidate methods of dealing with uncertainty, with increasing sophistication, in §5.3, §5.4 and §5.5.

For the purposes of this section, we assume that we do not use negation in hQuery. Even with this assumption, the issues that come up when dealing with uncertainty and bias are non-trivial.

5.1 Sources of Uncertainty

In the previous section, the only source of uncertainty is when not all certain answers are obtained. Additional tuples may or may not be present in the query result that is returned to the user — as a result of this uncertainty, the resulting answer could be one of several possible worlds [8], where the worlds contain all the certain answer tuples obtained, and may or may not contain the remaining answer tuples.

Also, in §4, we assumed that each human is omniscient; and

answers every task correctly. It is therefore sufficient to ask a single human to evaluate any h-predicate on a given tuple via the CrSS. However, the reality is very different.

Even for a unit task (i.e., a single question), there can be uncertainty in three ways: (a) Humans may make mistakes, e.g., Count the number of objects in an Image. (b) Humans have subjective views, e.g., Is the location depicted in this image clean? (c) Humans may provide spam/malicious responses. In this section, we consider (a) and (b), and we consider (c) in §6.2. For the purposes of this section, we assume that humans do not maliciously provide incorrect answers.

When there are multiple h-predicates, operating on multiple tuples/objects, there could be further sources of uncertainty or bias: (a) Answers that humans give to various h-predicates for the same tuple/object may be correlated. For instance, if a human thinks that an image is that of a city, then his answer to `hCity` would be (negatively) correlated with that of `hBeach`. (b) Answers that humans give to different tuples for the same h-predicate may be correlated. (c) Number of workers attempting each question may be very different, as a result, we may have several YES/NO answers on some questions for a given predicate but may not have them on other questions for the same predicate. (d) Number of people attempting different h-predicates may be different.

Thus, since the provided answers may be biased, incomplete, unnormalized or uncertain, we need to develop the means to reason when a h-predicate is true (on a specific tuple) given the answers of human workers for the same predicate and other predicates. Uncertainty also arises when dealing with ha-predicates or a-predicates. To handle this case, we assume that algorithms will provide an estimate on the reliability of the generated answers, which will allow us to handle answers from the CrSS and algorithms in a unified framework.

5.2 Incorporating Uncertainty

Since we have uncertainty as to whether or not an actual tuple is part of the query result, we can resort to using probabilities to quantify our belief as to whether or not a tuple is correct. This uncertainty could be in two forms, the probability of presence or absence of a tuple (defined before in uncertain databases [36, 12, 9] as *maybe* annotations), or the probability of each of a set of values for an attribute or group of attributes in a tuple (defined in uncertain databases as *alternative* annotations).

As a natural extension of *certain answers*, we propose the use of *probability thresholds*: For each way of using hQuery as defined in §3, we are given an additional parameter τ, representing the probability threshold. In other words, only tuples whose probability of being present in the query result is $\geq \tau$, are correct and should be returned to the user.

For instance, for problem 1, instead of requiring only certain answers to be present in the query result as in the previous section, we would like all tuples whose confidence is $\geq \tau$ to be present in the query result. Notice that when $\tau = 1$, this is equivalent to enforing all and only certain answers to be present in the query result, thus our technique of incorporating uncertainty into the ways of using hQuery forms a natural generalization of the ways of using hQuery in §3 when answers are uncertain.

Thus the main challenge that we will deal with in subsequent sections is how we compute confidences of tuples derived from h-ha- and a-predicates, and how we combine these confidences to compute those of tuples that should be in the query result.

5.3 Majority Voting: The easy way out

For the first method of computing probabilities, we use the simple strategy of majority voting. This scheme assumes that a fixed (large) number of identical, but independent workers attempt each question. In addition, no worker attempts more than one question. Thus, there are very few biases that come into play in this setting.

We compute the resulting answer for a h-predicate on a given tuple as simply the majority of the answers of the humans. Indeed, there is an implicit belief that the majority would be a good proxy for the correct answer. It is easy to see that this scheme essentially eliminates all probabilities, by simply awarding the most 'likely' alternative a probability of 1, while all other alternatives are awarded a probability of 0. Subsequently, we can produce results as we do for certain answers in §4.

Thus, in this scheme, it is not possible to get fine-grained probabilities for various tuples. As a result, there could be errors. For instance, we may miss out on certain tuples whose probabilities may be greater than τ, but were generated in conjunction with an alternative binding for a human-computed tuple that was not the one that had the majority. Secondly, when faced with alternatives that have nearly the same number of votes, or when there are many alternatives with the same number of votes, it is not clear how we can meaningful the majority answer is. One option is to simply ask more humans on the fly.

5.4 Probabilistic Approach with IID Workers

Here, we assume that for each question, workers are drawn independently and identically distributed (IID) from an a-priori distribution. For instance, there is an inherent distribution for the question `hBeach(Image3)`, which could be, for instance, Yes - 0.6, No - 0.4, and each human attempting the corresponding question would sample randomly from this inherent distribution. Of course, this distribution is not given to us, and needs to be inferred based on the answers of the humans to the question, based on maximum likelihood (i.e., each alternative has a probability equal to the fraction of humans that select it as an answer), or some other a-posteriori inference approach [11]. For hybrid predicates, we need to combine two distributions (one from algorithmic evaluation, and one from human computation) to give one unified distribution over alternatives.

This distribution is the same as distributions over alternatives in uncertain databases [36]. We assume, as in the previous scheme, that a fixed large number of people attempt each question. Thus, the probabilities are normalized and there are no additional biases.

Subsequently, for each unevaluated h-predicate on a given tuple, we assume that the probability is 0 for all alternatives. Once predicates get evaluated either by humans or by algorithms, the probabilities for various alternatives get updated. We can use standard confidence computation (as in uncertain databases) in order to compute confidence of each result tuple. These probabilities would have to be recomputed every time a human predicate is evaluated.

However, if there is no negation, then the confidence of each output tuple can only increase with additional h-predicate evaluation. We can therefore use this property to avoid evaluating additional human predicates that derive the same output tuple (that already has a confidence $\geq \tau$), and we can return output tuples as and when they become available. Of course, for those output tuples whose confidence is $< \tau$, it might still be the case that with additional human predicate evaluation, they become part of the query result.

5.5 Approach Using Item-Response Theory

In this scheme, we utilize all forms of uncertainty considered in §5.1. This situation has parallels with questionnaire analysis, when a questionnaire containing several questions is attempted by

many humans. Each human may choose to answer some or all of the questions. As in our case, there may be correlations between answers to various questions for each human, as well as uncertainty in the answers for even a single question.

Questionnaires are usually analyzed using Item Response Theory (IRT) [2]. Item Response Theory is the science of inferences and scoring for tests, questionnaires and examinations. In our case, the inferences may be especially tricky given that only a handful of humans attempt each question.

6. EXTENSIONS

We now describe some interesting extensions to the basic model.

6.1 Costing

An important aspect of posting tasks to the CrSS is to decide how to price each task. So far, we have assumed that we price each task equally, as a result of which the number of tasks posted is a measure of total amount spent. However, we may be able to speed up query processing by intelligent pricing of tasks.

More specifically, it is clear that we can reduce latency by assigning higher prices. Additionally, for tasks that help us obtain query tuples easier, it would be helpful to price them higher (for instance, if there are two queries deriving output tuples, one with a single h-predicate, and the other with 4 h-predicates, it would be beneficial to price the single h-predicate higher, since that would yield query tuples faster). Also, for tasks that are harder to attempt, pricing should be higher, otherwise humans may prefer to attempt other tasks earlier. Additionally, if we have a deadline for task completion, it would be wise to use a low price early in the evaluation process, and to increase the price closer to the completion deadline.

However, to maximize the number of questions under a fixed budget, we may want to replicate each question fewer times and issue a low price, at the cost of getting delayed, uncertain answers.

6.2 Spam

As in any setting, we need to safeguard our CrSS from adversarial attacks. These attacks could be similar to link farms (where a clique of nodes in a network gets a high pagerank due to high connectivity amongst themselves [17]), as well as algorithms mimicking humans (in other words, spam).

There are two ways of beating spam (each with limitations): (a) Use majority voting, penalizing any human who does not produce the majority answer. However, this method is susceptible to groups of algorithms mimicking humans colluding together to answer questions arbitrarily in the same fashion, as in a link farm. (b) Have a "test" question, with any human who gets it incorrect not being able to attempt the actual question. This test is akin to CAPTCHAs [34] that are essentially turing tests to discern algorithms from humans. If the ratio of test to actual questions is high, then the cost to use the CrSS may be too high. On the other hand, if the ratio of test to actual questions is low, then the humans may manually solve the test question, and then arbitrarily guess for the rest of the questions.

6.3 Second Order Predicates

Instead of h-predicates, one could also use human-computable algorithms, where the algorithms take in a question budget, and a latency requirement, select several questions to post to the CrSS and return an aggregated answer. These human-computable algorithms could be of various kinds, e.g., sort ten images, pick the odd one out, or cluster search results. For instance, the human-computable algorithm for sorting would be optimized to ask much fewer than the $\binom{n}{2}$ comparisons required. Design of a human computable algorithm to select an optimal set of questions to ask humans to solve a graph search problem has been examined in prior work [30].

6.4 Mapping to Unit Tasks

The choice of unit tasks corresponding to an h-predicate can affect the amount of time taken to obtain a result from the CrSS, the cost, as well as the inherent uncertainty in the answer. It is therefore an important dimension for query optimization. For instance, for a sorting predicate, we may use pairwise comparisons as our unit tasks (i.e., which of these two items is better?), or multi-way comparisons (i.e., what is the correct ranking for the following k items?). While pairwise comparisons may yield more accurate results, multi-way comparisons may result in answers being obtained faster, and at lesser cost. On the other hand, sorting by rating each item (i.e., how would you rate this item on a scale of 1-10?) would involve fewer tasks being posed to the CrSS, but may result in much higher uncertainty. Thus, in addition to deciding which questions to ask, the query optimizer needs to decide how to ask them to balance cost, uncertainty as well as latency.

6.5 Scheduling

In this section, we briefly discuss issues in the design of a query engine processing several hQuery queries in a batch. Firstly, we would want to schedule tasks at times based on availability of capable workers, e.g., if the query is to identify cities in Europe, it is pragmatic to schedule those tasks when it is daytime in Europe. Additionally, we may want to schedule and price tasks based on priority order of completion, i.e., price important tasks higher and post them earlier. As in standard joint query optimization, it would be beneficial to share computation between queries in the workload. In particular, h- and a-predicates that have been evaluated do not need to be reevaluated, and can be stored in an extensional database for reuse while processing other queries.

7. RELATED WORK

A number of recent works have considered the design of libraries to access crowd sourcing services such as Mechanical Turk [3]. Little et. al. [25, 26] suggested iterative and parallel tasks as design patterns in order to build more complex applications over Mechanical Turk. Heymann et. al. [20] describes the design of a programming environment built on top of Mechanical Turk that contains modular and reusable components.

Kochhar et. al. [24] describes how Freebase has been using human judgement for various data cleansing tasks. Crowd-sourcing services are also used to provide input to active learning algorithms [33]. A few works have also explored other aspects of crowd-sourcing, such as designing good tasks [22], pricing tasks [28, 21], the occurrence of spam and bias [23, 27] and so on. The survey by Doan et. al. [16] provides a good overview of the general area of human involvement or collaboration and how it may be used to solve problems.

We argued earlier about the necessity of adaptive query processing due to the difficulty of obtaining accurate cost models for h- and a-predicates. We hope to leverage the extensive previous work on this topic in the context of relational database systems [15]. Additionally, the issues of uncertainty arising in human computable predicates are more intricate than the ones found in existing uncertain databases [36, 9, 12], however some of the confidence computation techniques therein can be reused in our context.

There has been some work in the past on optimizing query plans that contain expensive predicates [14, 19], however, in our case, we also need to deal with pricing as well as uncertainty. The work

on online aggregation [18] allows users to specify on the fly which data is important and needs relational processing. However, in our case, humans perform the processing.

Additionally, from an application standpoint, there have been several papers considering the use of human experts in various tasks such as data integration and exploration [31, 35, 32, 29] and information extraction [13]. Our previous work examined the optimization of graph search tasks that harness human processors via a crowd-sourcing service [30].

8. CONCLUSIONS

With the vision of being able to process and answer queries on data more effectively, we proposed, in this paper, the use of a CrSS in addition to standard relational database operators. We presented hQuery, a declarative query model operating over human-computable predicates, databases as well as external algorithms, and discussed the challenges involved in optimizing hQuery, in particular, the presence of uncertainty derived from the answers of humans, as well as tradeoffs between number of certain (or probabilistic) answers, time allocated, and monetary cost. In addition, due to not-so-well-understood operators, unknown latencies, accuracy and selectivity and lack of cardinality estimates, query processing can become especially hairy. We defined several research problems, with increasing levels of complexity, for the case of perfect answers, as well as when answers are uncertain. This work also raises some research issues on how humans interpret and answer questions. We believe that the research problems outlined herein, and the general area of optimizing humans as well as databases, form an interesting and important area for future database research.

9. REFERENCES

[1] CrowdFlower. *http://crowdflower.com*.

[2] Item Response Theory. *http://en.wikipedia.org/wiki/Item_response_theory*.

[3] Mechanical Turk. *http://mturk.com*.

[4] Microtask. *http://microtask.com*.

[5] ODesk. *http://odesk.com*.

[6] Samasource. *http://samasource.com*.

[7] UTest. *http://utest.com*.

[8] Serge Abiteboul, Paris Kanellakis, and Gosta Grahne. On the representation and querying of sets of possible worlds. *SIGMOD Rec.*, 16(3):34–48, 1987.

[9] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions. In *Proc. of ICDE*, 2007.

[10] Jeff Barr and Luis Felipe Cabrera. Ai gets a brain. *Queue*, 4(4):24–29, 2006.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1st ed. 2006. corr. 2nd printing edition, October 2007.

[12] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proc. of ACM SIGMOD*, 2005.

[13] Xiaoyong Chai, Ba Q. Vuong, Anhai Doan, and Jeffrey F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD '09*.

[14] Surajit Chaudhuri and Kyuseok Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.

[15] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[16] AnHai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Mass Collaboration Systems on the World-Wide Web. Technical report, Communications of the ACM (To Appear).

[17] Ye Du, Yaoyun Shi, and Xin Zhao. Using spam farm to boost pagerank. In *AIRWeb '07*, New York, NY, USA, 2007.

[18] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD Conference*, 1997.

[19] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD Conference*, pages 267–276, 1993.

[20] Paul Heymann and Hector Garcia-Molina. Human processing. Technical report, Stanford University, July 2010.

[21] John Horton and Lydia Chilton. The labor economics of paid crowdsourcing. *CoRR*, abs/1001.0627, 2010.

[22] Eric Huang, Haoqi Zhang, David C. Parkes, Krzysztof Z. Gajos, and Yiling Chen. Toward automatic task design: a progress report. In *HCOMP '10*, New York, NY, USA, 2010.

[23] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *HCOMP '10*, New York, NY, USA, 2010.

[24] Shailesh Kochhar, Stefano Mazzocchi, and Praveen Paritosh. The anatomy of a large-scale human computation engine. In *HCOMP '10*, New York, NY, USA, 2010.

[25] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP '09*, New York, NY, USA, 2009.

[26] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Exploring iterative and parallel human computation processes. In *HCOMP '10*, New York, NY, USA, 2010.

[27] M. Motoyama et. al. Re: Captchas understanding captcha-solving services in an economic context. In *USENIX Security Symposium '10*.

[28] Winter Mason and Duncan J. Watts. Financial incentives and the "performance of crowds". In *HCOMP '09*, New York, NY, USA, 2009.

[29] Robert McCann, Warren Shen, and AnHai Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE '08*.

[30] Aditya Parameswaran, Anish Das Sarma, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Human-assisted graph search: It's okay to ask questions. Technical report, Stanford University.

[31] S. Jeffery et. al. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD '08*.

[32] S. Sarawagi et. al. Interactive deduplication using active learning. In *KDD '02*.

[33] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[34] Luis von Ahn and Laura Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8), 2008.

[35] W. Wu et. al. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD '04*.

[36] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Proc. of CIDR*, 2005.

The Case for Predictive Database Systems: Opportunities and Challenges

Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, Stan Zdonik
Department of Computer Science
Brown University, Providence, RI, USA
{makdere, ugur, matteo, eli, sbz}@cs.brown.edu

ABSTRACT

This paper argues that next generation database management systems should incorporate a predictive model management component to effectively support both inward-facing applications, such as self management, and user-facing applications such as data-driven predictive analytics. We draw an analogy between model management and data management functionality and discuss how model management can leverage profiling, physical design and query optimization techniques, as well as the pertinent challenges. We then describe the early design and architecture of **Longview**, a predictive DBMS prototype that we are building at Brown, along with a case study of how models can be used to predict query execution performance.

1. INTRODUCTION

Predictive modeling has been used with varying degrees of success for many years [GH05]. As models grow more sophisticated, and data collection and storage become increasingly more extensive and accurate, the quality of predictions improves. As such, model-based, data-driven prediction is fast emerging as an essential ingredient of both user-facing applications, such as predictive analytics, and system-facing applications such as autonomic computing and self management.

At present, predictive applications are not well supported by database systems, despite their growing prevalence and importance. Most prediction functionality is provided outside the database system by specialized prediction software, which uses the DBMS primarily as a backend data server. Some commercial database systems (e.g., the data mining tools for Oracle [Ora], SQL Server [SS08], and DB2 [DB2]) provide basic extensions that facilitate the execution of predictive models on database tables in a manner similar to stored procedures. As we discuss below, and also noted by others (e.g., [DB07, AM06]), this loose coupling misses significant opportunities for improved performance and usability. There has also been recent work on custom integration of specific models (e.g., [JXW08, HR07, ACU10, AU07, APC08]).

This paper argues that next generation database systems should natively support and manage predictive models, tightly integrating them in the process of data management and query processing. We make the case that such a **Predictive Database Management System (PDBMS)** is the natural progression beyond the current afterthought or specialized approaches. We outline the potential performance and usability advantages that PDBMSs offer, along with the research challenges that need to be tackled when realizing them.

A PDBMS enables **declarative predictive queries**—by providing predictive capability in the context of a declarative language like SQL; users will not need to concern themselves with the details of tasks like model training and selection. Such tasks will be performed by the optimizer behind the scenes, optionally using hints from the user. Much as SQL has made programmers more productive in the context of data processing, this approach will have a similar effect for predictive analytics tasks. While there will no doubt be some predictive applications that can benefit from custom, manually optimized prediction logic, we expect that many users will be satisfied with "commodity" predictive functionality. The success of the recent Google Prediction API [GP] is early evidence in this direction. This service allows users to upload their historical data to the service, which automatically and transparently performs model training and selection to produce predicted results.

Predictive queries have a broad range of uses. First, they can support predictive analytics to answer complex questions involving missing or future values, correlations, and trends, which can be used to identify opportunities or threats (e.g., forecasting stock-price trends, identifying promising sponsor candidates, predicting future sales, monitoring intrusions and performance anomalies).

Second, predictive functionality can help build introspective services that assist in various data and resource management and optimization tasks. Today, many systems either use very simple, mostly static predictive techniques or do not use any prediction at all. This is primarily due to the difficulty of acquiring the appropriate statistics and efficiently and confidently predicting over them. For example, pre-fetching algorithms are often based on simple linear correlations to decide future data or query requests. Most admission control schemes are based on static estimations (thresholds) of the maximum number of tasks that the system can cope with. Load distribution algorithms typically detect-and-react instead of predict-and prevent problems. Query optimizers commonly use simplistic analytical models to reason about query costs. There is major recent interest and success in applying sophisticated statistical and learning models to such problems [GKD09, BBD09, SBC06]. An integrated, readily available predictive functionality would make it easy to not only consolidate and replace existing solutions but also build new ones. As such, an integrated predictive functionality would be an

5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

important step towards building the truly autonomic database systems of the future.

A PDBMS integrates predictive models as first-class entities, managing them in much the same way as data. Thus, we consider **model management** as the key underlying component of a PDBMS. Model management may greatly benefit from analogues of many well-established data management techniques:

- **Profiling and modeling:** Cost and accuracy characteristics of models need to be modeled, and fed to the optimizer so that the proper model(s) can be chosen for a given predictive task.
- **Physical design and specialized data structures:** Data can be structured to facilitate efficient model building and predictive query execution (e.g., I/O-aware skip-lists [GZ08]).
- **Pre-computation and materialization:** Model building is often prohibitively expensive for ad hoc or interactive queries. In such cases, models can be pre-built and materialized for use by the optimizer and executor. Furthermore, this process can be automated in many cases.
- **Query optimization:** The optimizer considers the alternative ways of model building, selection, and execution, as well as the inherent cost-accuracy tradeoffs when selecting an execution plan.

In the rest of the paper, we discuss these model management techniques as well as the technical challenges that arise when building a PDBMS. Our discussion is centered on **Longview,** a prototype predictive DBMS that we have been building at Brown University. Longview is being designed to efficiently support declarative predictive analytics through novel integrated model management techniques. Users can plug new model types into the system along with a modest amount of meta-data, and the system uses these models to efficiently evaluate queries involving predictions.

We sketch the basic architecture of Longview and its early implementation on top of PostgreSQL. We also discuss an internal predictive application, query performance prediction, which exercises some of the model management issues we raise. Finally, we discuss prior work and finish with concluding remarks.

2. BACKGROUND: PREDICTION WITH MODELS

We use the term model to refer to any predictive function such as Multiple Regression, Bayesian Nets, and Support Vector Machines. Training a model involves using one or more data sets to determine the best model instance that explains the data. For example, fitting a function to a time series may yield a specific polynomial instance that can be used to predict future values.

In general, **model training** (or **building**) involves selecting (i) the feature attributes, a subset of all attributes in the data set, and (ii) a training data set. In some cases, a domain expert can manually specify the feature attributes. In other cases, this step is trivial as the prediction attribute(s) directly determine the feature attribute(s), e.g., as in the case of auto-regressive models. Alternatively, feature attributes can be learned automatically. Most solutions for automatic learning are based on heuristics, since given a set of n attributes, trying the power set is prohibitively expensive if n is not small or training is costly [GH05, MWH98]. A common approach is to rank the candidate attributes (often based on their correlation to the prediction attribute using metrics such as information gain or correlation coefficients [CT06]) and use this ranking to guide a heuristic search [GH05] to identify the most predictive attributes tested over a disjoint test data set. The training data set may be sampled to speed up the process.

Prediction **accuracy** is a function of the quality of the estimated models. The quality of the model (and the resulting predictions) can be measured by metrics such as the variation distance [MU05] or the mean square error between the predictions and the true values. With assumptions about the underlying stochastic process, one may be able to bound these measures analytically, using large deviation theory, appropriate versions of the central limit theorem and martingale convergence bounds [MU05]. Alternatively, one can use multiple tests on available data to compute the empirical values for these measures. However, using empirical values to estimate the model or prediction error adds another layer of error to the estimate, namely the gap between the empirical statistics and the true value they estimate. While the empirical statistic is an unbiased estimate, the variance of the estimate can be large, depending on the size and variance of the test set.

Hypothesis testing and confidence interval estimations are two common techniques for determining predictive accuracy [MWH98]. In general, it is not possible to estimate a priori what model would be most predictive for a given data set without training and testing it. One form of hypothesis testing that is commonly used is K-Fold Cross Validation (K-CV). K-CV divides up the training data into k non-overlapping partitions. One of the partitions is used as validation data while the other k-1 partitions are used to train the model.

3. LONGVIEW: A PREDICTIVE DBMS

3.1 Design and Architecture Overview

3.1.1 Data and Query Model

Longview provides two interfaces for access to its predictive functionality. The first access method is the **direct interface**, which consists of a collection of SQL functions that offers direct access to the functionality of the integrated prediction models. The direct interface does not provide the user with automated model management tools and is thus targeted towards advanced users who want to exert hands-on control on the prediction models and their operations. For example, using this interface a user can ask the system to build a linear regression model with specific configuration parameters or perform prediction with a pre-built support vector machine instance. We summarize the details of the direct interface in Section 3.2.1.

The second access method is the **declarative interface**, which offers additional, high-level predictive functionality on top of the low-level direct interface.

This declarative interface extends SQL in a few simple ways to accommodate the extra specifications needed for expressing predictive queries. In particular, queries may refer to **predictors** and **predictor relations** (**p-relations**) to access predictive functionality. Predictors are essentially SQL functions that provide declarative predictive functionality using system-managed prediction models. P-relations are essentially views produced by the application of predictors on select subsets of input features. Both p-relations and predictors can be used in conjunction with regular relations within standard SQL queries. A

p-relation is virtual by default; however, it can also be materialized to enable further optimizations.

We give a simple example that illustrates some of the key concepts of the query language that we are developing. Consider the following schema:

```
Customer(cid, name, city),
Orders(oid, cid, total),
TrainData(cid, status)
```

In addition to the Customer and Orders relations, which store the records for customers and their orders, we define the TrainData relation that stores the status (either "preferred" or "regular") of a subset of the customers. We first show how to build a predictor for predicting the status of any customer based on the training data supplied for a subset of the customers. Next, we discuss p-relations and their use through an example p-relation representing the status predictions of a select subset of customers based on a predictor.

The first step in creating a predictor is to define a schema describing the set of involved features and target attributes. For this purpose, we define a schema, named StatusSchema, with the target attribute customer status and features name, city and total using the CREATE P_SCHEMA statement:

```
CREATE P_SCHEMA StatusSchema (
        name text,
        city text,
        total int,
        TARGET status text)
```

To create a predictor, we use the CREATE PREDICTOR statement that can be used to automatically build prediction model(s) using the given training data set:

```
CREATE PREDICTOR StatusPredictor
ON StatusSchema(name, city, total, status)
WITH DATA
        SELECT name, city, sum(total) as total, status
        FROM Customer C, Orders O, TrainData T
        WHERE T.cid = C.cid and T.cid = O.cid
        GROUPBY cid, name, city, status
WITH ERROR CVERROR(10, "relative_error", 0.1)
```

With the statement shown above, we instruct the system to create a predictor named StatusPredictor by training a set of prediction models using the training data specified through a query. The last part, WITH ERROR, defines the error estimation process. In this example, we want to use 10-fold cross-validation and the relative_error accuracy metric with a target average error of 0.1. Notice that the decoupling between the schema and predictor definitions allows us to create multiple predictors with different data sets or accuracy requirements over a single schema.

The example query below illustrates the use of the StatusPredictor for estimating the status of all customers:

```
SELECT C.cid, StatusPredictor(C.name, C.city, O.total)
FROM Customer C,
        (select cid, sum(total) as total from Orders
         Group By Cid) as O
WHERE C.cid = O.cid
```

The output schema of a predictor is defined by the associated p_schema. In addition, one can add special ERROR attributes to a p_schema to access the estimated errors for each predicted value. For instance, adding the attribute "ERROR relerr real" to p_schema would extend the output schema of a predictor with the *relerr* attribute, which represents the estimated prediction error.

Now, we describe how to define p-relations with the following example:

```
CREATE VIEW StatusPRelation AS
SELECT cid, StatusPredictor(name,city,total)
FROM (
    SELECT cid, name, city, sum(total) as total
    FROM Customer C, Orders O
    WHERE C.cid = O.cid
    GROUP BY cid, name, city, status
    HAVING sum(total) > 1000)
```

With the above statement, we create a p-relation named StatusPRelation, which is basically a view consisting of status predictions from StatusPredictor for the set of features specified with the provided query (i.e., customers with order totals greater than 1000) and the features themselves.

When a view definition that accesses a predictor function is submitted to the system, Longview registers the given data set as a specific target feature set for that predictor. In turn, the model generation process for the predictor works to generate more efficient and accurate prediction models based on the properties of the given feature set.

The use of declarative queries for the specification of data sets in model building and prediction offers an easy and flexible method of expressing predictive operations over complex data sets. Users can easily specify complex queries (e.g., computing aggregates over groups) to supply input data sets for prediction models. Moreover, it is also possible to use database views as data providers. For instance, a database view can be used to perform standard pre-processing tasks such as cleaning, normalization, and discretization [DB07], and can cook the raw data into a form that is more amenable for effective learning.

3.1.2 Basic Architecture

We illustrate the high-level architecture of Longview in Figure 1, which shows the primary functional units of interest, along with the data they require. The architecture reflects the notion of models as first-class citizens by depicting the data manager and model manager as co-equal modules.

The Data Manager is very similar to a typical data manager in a conventional DBMS. The Model Manager is responsible for creating materialized models a priori (materialized) or in an on-demand fashion when an adequate materialized model does not exist. The role of materialized models in the model world is similar to that of indices and materialized views in the data world: they are derived products that can be used to quickly generate data of interest. Indices and materialized views improve query speed while materialized models improve prediction speed.

The Model Manager trains appropriate models (based on the available model templates) for each predictor in the database. The Model Manager can run as a background process, constantly instantiating models for improved accuracy and efficiency. In order to build and maintain prediction models, the Model

Manager can utilize many different strategies. For example, it can choose to sample data at different amounts and times and it can build different types of prediction models over different subsets of available features. In addition, the Model Manager also determines the best model for the query at hand: if an appropriate model has already been pre-computed and materialized, it will identify and use that model; if not, it will create a new instantiation on the fly.

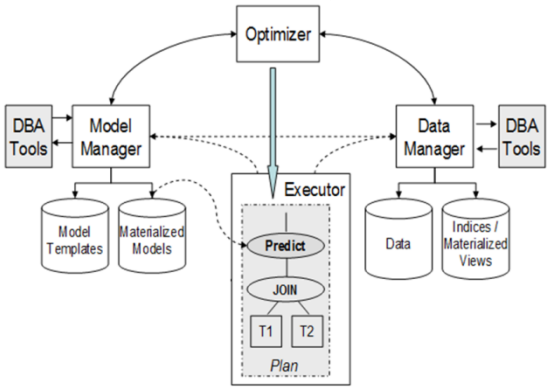


Figure 1: High-level Longview architecture. The system provides full-fledged support for models; model and data management are tightly integrated.

The Model Manager and the Data Manager must cooperate in their decision making. As we discuss later, special data structures can assist the process of model training. This is consistent with the fact that DBMSs, in general, get much of their performance gains from supporting specialized data structures like indices.

Model meta-data entered through the model interface as well as those derived during run-time such as the list of materialized models and their parameters, training results, error values for various data sets, are all stored in a model catalog. The model manager is responsible for updating the catalog.

Longview will try to produce a good model whenever possible by trying various parameter assignments (e.g., history length, sampling density, etc.) and using hypothesis testing to find the best fit. While Longview aggressively tries to optimize this model search process, in some cases, this is either not possible or would require testing too many alternatives. In these cases, Longview will provide a set of tools with which the DBA can inspect the data and add additional information about the datasets which might indicate, for example, that the data is seasonal, or that the data might best be modeled using exponential smoothing. Traditional DBMSs provide such tuning tools for DBAs as well.

P-relation queries are written against views that include predicted attributes. When a p-relation query is received by the system, the optimizer might generate a query plan that contains prediction operators. These operators are selected from a collection of instantiated models that are managed by the Model Manager, or created on the fly. Alternatively, tuples in the predicted view can be computed eagerly and materialized as resources become available, in which case p-relation queries can be executed as scans over the materialized tuples.

3.2 Model Management

As a design philosophy towards a generic model manager, we strive to build on existing database extension mechanisms such as views, triggers, rules and user defined functions to simplify our implementation and produce highly portable functionality.

3.2.1 Database Integration of Prediction Models

Prediction Model API. Longview currently supports a black-box-style integration approach that allows existing model implementations (available from a plethora of standalone applications and libraries such as libsvm [CC01]) to be used by the system as database functions. This approach offers an easy and effective way of utilizing pre-tested and optimized prediction logic within a SQL execution framework. New prediction models are registered into the system by providing implementations of a simple model interface (the prediction model API) describing function templates for training and application of prediction methods. This interface decouples implementation and predictive functionality, while allowing multiple predictive models to be used for the same task. Table 1 summarizes the basic interface methods.

Function	Arguments	Description
Build	*training data* *model parameters*	feature and target values model-specific training parameters
Predict	*model pointer* *feature list*	pointer to previously built model feature values for use in prediction
Serialize	*model pointer*	
Deserialize	*byte array*	serialized model

Table 1 - Prediction Model API

The build function is used to train a prediction model based on the given features and target values, as well as model-specific training parameters. The predict function uses a previously built model to predict a target attribute based on the input feature values. Finally, Longview uses the serialize and de-serialize functions to store and retrieve prediction models. Most third-party model libraries include built-in model (de)serialization methods for this purpose.

Prediction Model Direct Interface. The prediction model API is used internally by the Longview system to access the functionality of prediction models and is not visible to the user. However, as mentioned earlier, Longview also provides an interface for direct access to the prediction models by the user. The main functions included in this interface are given in Table 2. These functions have dynamic implementations in Longview, as wrappers around the prediction model API, and provide a unified method of access to all the available prediction model types within SQL statements.

The create function is used to create a prediction model entry in the model catalogs for the given model type and attribute schema. The Longview model catalog stores all model data and associated meta-data. Each model instance built is recorded in a relation that contains a unique (auto-generated) instance id, model type, and a serialization field storing the type-specific representation of a prediction model (e.g., the coefficients of a regression model). We also store model attributes; each is represented with a name, id, a type (e.g., double) and a role (i.e., feature, target).

The build and predict SQL-functions are similar to the corresponding functions in the prediction model API. The build

function trains the prediction model specified by the model id, and stores its serialized representation in the model catalog. The predict function performs prediction with the given model id over the provided feature data set. We also provide a test function that can be used to apply the model on a feature data set and compute its accuracy over the true values of the target attributes. We provide an argument to specify the accuracy function for use (e.g., absolute error, squared error). The outputs of the test and prediction functions are represented as relations and can be used as data sources in other queries.

Function	Arguments	Description
Create	*model schema* *model type*	description of features and target attributes prediction model type
Build	*model id* *training query* *model parameters*	specifies the model instance query computing the feature and target values model-specific training parameters
Predict	*model id* *feature list | query*	feature values for use in prediction
Test	*model id* *training query* *accuracy options*	parameters for the accuracy function

Table 2 - Prediction Model Direct User Interface

3.2.2 Model Building and Maintenance

Model Materialization. Longview builds and materializes model instances much as a conventional DBMS pre-computes indices or materialized views. For each predictor and associated p-relations, there can be multiple materialized prediction models built using different model types and different feature subsets. As a result, model building and maintenance may easily become a bottleneck as the number of pre-built models increases. Therefore, methods for decreasing the cost of building and maintaining models are an essential part of Longview.

The quality of a model is primarily a function of its training data and model-specific configuration parameters. In the limit, we would like to produce one materialized model for each prediction query. This approach will likely be infeasible for two reasons: (1) the time required to build a model per query is larger than some target threshold, e.g., in applications involving interactive queries; and (2) the estimated time required to update these models in the face of newly arriving data is greater than some maintenance threshold.

In many ways, this problem is very similar to the problem of automatic index or materialized view selection. We require (1) a reasonably good cost and accuracy model that can be used to compare the utility of the materialized models, and (2) a way to heuristically prune the large space of possible models.

A good solution to this problem involves covering the underlying "feature space" well such that a prediction with acceptable accuracy can be made for a large set of queries subject to a limit on model maintenance costs. In prior work, we proposed a solution along these lines for time-series-based forecasting using multi-variate regression [GZ08].

In addition to the techniques mentioned earlier such as sampling, feature selection and materialized models, there are further opportunities to reduce the execution costs of these tasks. First, these operations can be done in parallel for multiple models on the same data. In this **multi-model building** process (akin to multi-query optimization), data can be read once and all relevant models can be updated at the same time. Moreover, we can build and update models in an **opportunistic** manner based on memory-resident data.

Auto Design. The auto-design problem is a related problem in which the goal is to choose and build a set of prediction models based on a given workload that contains a set of predictive queries that are most likely to be submitted, i.e., queries that we would like to execute quickly and with good predictive accuracy. For this purpose, the database system would need to identify the most common prediction attributes in the workload and then the set of features that are highly predictive of those attributes.

Specialized Data Structures. There are opportunities for a PDBMS to leverage data representations that are tuned to the process of prediction. In particular, structures that can enhance model training have the most potential to yield major performance improvements with the idea being accessing "just enough" data to build a model of acceptable accuracy.

Data-driven training commonly involves accessing select regions in the underlying feature space, combined with sampling techniques that can be used to further reduce I/O requirements. This process is often iterative: more data is systematically included to check if the resulting model is better. In general, multi-dimensional index structures defined over the feature space can be effectively used here, but care must be taken that index-based sampling does not introduce any biases. Multi-dimensional clustering, when performed in a manner that facilitates efficient sampling, can provide further benefits. As an alternative to the index-based sampling of disk-resident data, we can also opt to replicate the data (or materialize the results of a training query) using disk organizations tuned for efficient sampling, e.g., horizontally partition the data into uniform samples so that sampling can be done with sequential I/O.

As a concrete example for time-series prediction, we introduced a variant of skip lists to efficiently access arbitrary ranges of the underlying time dimension with different sampling granularities.

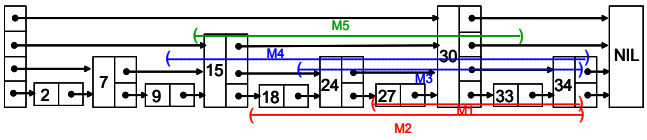


Figure 2: I/O conscious skip-lists. Each node indicates a block of tuples sampled from the original relation. Unlike in standard skip lists, nodes (blocks) are not shared across levels. M's indicate different time ranges and sampling density.

The original skip-list formulation is modified to make it I/O conscious by copying the relevant data from each lower level up to the higher-levels. Each level is essentially a materialized sample view [JJ08] stored in clustered form on disk, allowing us to access a particular time range with desired density with a small number of disk accesses (see Figure 2 for an illustration).

3.2.3 Query Execution and Optimization

Predictor optimization. Declarative predictive queries specify what to predict but not how. For a given prediction task, it is the responsibility of the predictor to build and use an appropriate prediction model satisfying the desired accuracy. For this purpose, each Longview predictor continuously tries to build accurate prediction models for as much of its input feature space as possible, while keeping resource consumption under a configurable threshold to avoid negatively impacting the other database tasks. In the case of p-relations, predictors can build more targeted prediction models using select parts of the training data (i.e., **model segmentation**) based on the target data of p-relations. We discuss an application of the model segmentation idea and demonstrate its potential in Section 4.

In addition, Longview automatically keeps track of model cost-accuracy characteristics. For each model instance, the run-time cost and quality of the predictions during build and test operations are recorded. Using this information, Longview can monitor the evolution of models, track the used training data sets and the performance values on test data sets. These **model profiles** guide query optimization decisions. We may also expect expert users (or model developers) to supply simple cost functions, akin to those for the relational operators, for training and prediction costs, which can also be stored and leveraged as part of model profiles.

Finally, we observed the need for a more formal, expressive tool when working with sophisticated prediction models. To this end, we believe that a **model algebra** that captures common model operations such as choice (selection), composition, and merge is warranted. Properties of these operations could introduce further functionality as well as optimization opportunities. A model ensemble, which uses a set of prediction models collectively to perform a prediction task, is an example for this complex model case. Model ensembles rely on the collective power of multiple prediction models to smooth their predictions and mitigate the potential errors from a single prediction model.

Online execution. Online execution of predictive queries (along the lines of online aggregation), in which predictions, and thus the query results, get progressively better over time, is an important usage model for interactive, exploratory tasks. Predictive accuracy can be improved over time using more data, more features, or more models. The challenge is to effectively orchestrate this process and perform efficient revision of query results.

4. CASE STUDY: PREDICTING QUERY EXECUTION LATENCIES

We now describe our ongoing work on an inward-looking predictive task, **query performance prediction** (QPP), which involves the estimation of the execution latency of query plans on a given hardware platform. Modern database systems can greatly benefit from accurate QPP. For example, resource managers can utilize QPP to allocate workload such that interactive behavior is achieved or specific quality of service targets are met. Optimizers can choose among alternative plans based on expected execution latency instead of total work incurred.

While accurate QPP is important, it is also challenging: database systems are becoming increasingly complex, with several database and operating system components interacting in sophisticated and often unexpected ways. Analytical cost models are not designed to capture these interactions and complexity. As such, while they do a good job of comparing the costs of alternative query plans, they are poor predictors of plan execution [GKD09].

As an alternative, we express the QPP task using the declarative prediction interface in Longview. In addition to describing the query specification and execution, we also show different modeling approaches to achieve accurate QPP under various workload scenarios. If a representative workload is available, for example, we can build good models using coarse-grained, plan-level models. Such models, however, do not generalize well, and perform poorly for unseen or changing workloads. In these cases, fine-grained, operator-level modeling performs much better due to its ability to capture the behavior of arbitrary plans, although they do not perform as well as plan-level models for fixed workloads. We then build hybrid models that combine plan- and operator-level models to provide the best of both worlds by striking a good balance between generality and accuracy.

Plan-level Prediction. We first consider a basic approach that extracts features from query plans and then couples them with sample plan executions to build models using supervised learning (as also explored in [GKD09]). Once built, these models can perform predictions using only static plan information. The following features are extracted from each query plan for modeling purposes: optimizer estimates for query plan costs, number of output tuples and their average sizes (in bytes), and instance (i.e., occurrence) and cardinality counts for each operator type included in the query plan.

We integrated two prediction models, Support Vector Machines and Linear Regression, into the PostgreSQL database system (version 8.4.1) through the use of machine learning libraries LIBSVM [CC01] and Shark [IMT08]. We used the TPC-H decision support benchmark to generate our database and query workload. The database size is set to 10GB and experiments were run on a 2.4 GHz machine with 4GB memory. Our query workload consists of 500 TPC-H queries, which are generated from 18 TPC-H query templates and executed one after another with clean start (i.e., file system and database buffers are cleared).

Fixed Workload Experiment: In the first experiment, we defined a plan-level predictor using the described query plan features and the execution time target attribute as our p_schema (named *PlanSchema*). For this purpose, we first inserted the runtime query plan features and the execution times of all queries in our TPC-H workload to database tables (*runtimefeats* and *qexec*). Then, we defined our predictor to use 90% of the workload for building prediction models to estimate the execution times of the remaining 10% of the queries. We provide the definition of the plan-level predictor below; however at this point we do not have a SQL parser for the extensions proposed in the declarative predictor interface and thus performed our operations using the direct interface along with a few additional SQL functions that provide functionality similar to the declarative predictor interface.

```
CREATE PREDICTOR PlanLvlPredictor
ON PlanSchema(…)
WITH DATA
    SELECT R.*, Q.exec_time
    FROM runtimefeats R, qexec Q
    WHERE R.qid = Q.qid and R.qid <= 450
```

The *qid* attribute is a key in both tables that uniquely defines a query in the TPC-H workload. Next, we used the pre-runtime

estimations of the query plan features from the query optimizer (stored in table *estimatedfeats*) for performance prediction of the remaining 10% of the queries. The following query is used to express this operation:

```
SELECT qid, PlanLvlPredictor(…).exec_time
FROM estimatedfeats E
WHERE E.qid > 450
```

In this experiment, we used support vector machines (SVMs) as our prediction model. In addition, our current predictor optimizer uses a standard feature selection algorithm for choosing the set of features to use in prediction models. The set of features (7 of the total 29 features) used in the resulting model are: number of Group Aggregate, Hash Aggregate, and Materialization operators, estimated total plan cost, cardinality of Hash Aggregate and Hash Join operators and the estimated total number of output rows from all operators in the query plan. The error value (defined as |true value – estimate| / true value) for each TPC-H template is shown in Figure 3 (The average prediction accuracy is 90%).

We observed that queries from the 9th template (which has the unusual high errors) run close to the 1 hour time limit (after which we killed and discarded queries) and therefore execute longer than most other queries. We then performed manual model-segmentation by building a separate prediction model for the queries of the 9th template, which achieved 93% accuracy.

This example illustrates the potential efficiency of using segmented models built from different data partitions. As discussed before, intelligent model-building algorithms that automatically identify such partitions in the feature space are essential for improved accuracy.

Finally, when we added the additional feature used by that model (cardinality of the Nested Loop operator) to the general prediction model and retrained it, we increased its accuracy to 93% (shown with the bars for 2-step feature selection in the figure).

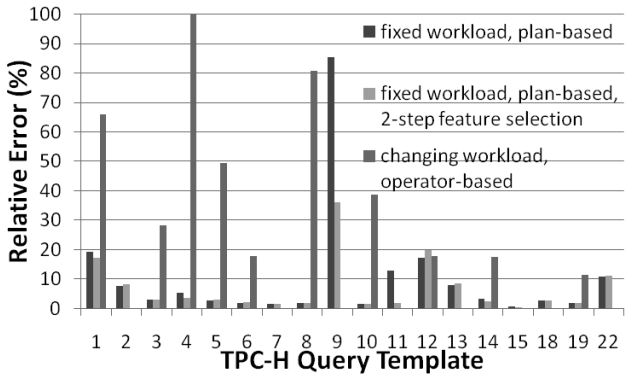


Figure 3: Query Prediction Performance for TPC-H Queries.

Changing Workload Experiment: In this experiment, we built separate prediction models for each TPC-H template using only the queries from the other TPC-H templates for training. In this case the average prediction error increased to 232%. In addition, the error values were highly dependent on the target query template and were distributed in a large range (2% to 1692%).

Operator-level Prediction. We also studied an operator-level modeling approach with the goal of building better models for the Changing Workload scenario. In this case, we build separate Linear Regression models to estimate the execution time for each *operator type* and compose them in a bottom-up manner up the query tree to predict the total execution time.

Each operator is modeled using a generic set of features such as the number of input/output tuples and estimated execution times for child operators (runtime and estimated values for these features are stored in *opruntimefeats* and *opestimatedfeats* tables). Bottom-up prediction requires a nested use of predictors. Moreover, the connections between predictors are dynamic as they depend on the plan of the query at hand. Currently, we perform this nested prediction operation within a user-defined database function that uses the operator predictors as required by the plan of each query. We think that such complex models can be built and used more effectively with a model algebra as mentioned in Section 3.

The results for the Changing Workload experiment using the operator-level prediction methods are shown in Figure 3 for 10 TPC-H templates. The average error rate is 56%, which represents a major improvement over query-level prediction for this workload.

Hybrid Prediction. Looking closer, we observe that the error (of 233%) for the operator-level prediction of template 4 queries is much higher than those for other templates. To gain more insight, we provide the error values for each operator in the execution plan for an example template-4 query (Figure 4). Observe that the errors originate from the highlighted sub-plan and propagate to the upper levels. Here, the error is due to the inability of the models to capture the per-tuple processing time of the Hash Aggregate operator, which in this case is computation-bound. Thus the I/O cost of the Sequence Scan operation that normally determines the overall execution latency is dominated by Hash Aggregate's high computational cost in this case. The fundamental problem is that operator-level training inherently fails to capture the "context" of the operator behavior.

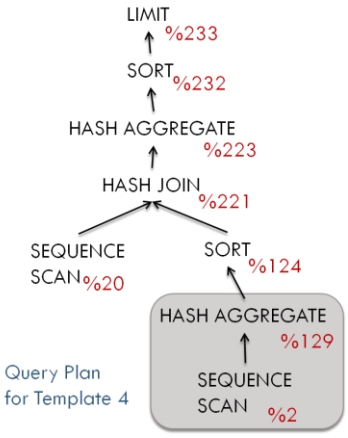


Figure 4: Query Tree and Prediction Errors for Template 4.

To solve this problem, we combined the plan- and operator-level prediction methods for template 4 by modeling the highlighted sub-plan with a plan-level model and using operator-level prediction for the remainder of the query plan. With this approach, we reduced the template error to 53% and the overall average error across templates to 38% for the Changing Workload scenario.

5. RELATED WORK

We draw from a large number of subject areas, which we summarize below. Other closely related work was cited inline as appropriate.

Major commercial DBMSs support predictive modeling tools (e.g., Oracle Data Mining tools, SQL Server Data Mining and DB2 Intelligent Miner). Such tools commonly allow users to invoke model instances, typically implemented as stored procedures, using extended SQL (e.g., the "FORECAST" clause [Ora]). In SQL Server Analysis Services, users are provided with a graphical interface within Visual Studio in which they can interactively build and use a number of prediction models such as decision trees and naïve Bayes networks. While we utilize similar prediction models and techniques, our goal is to create a more automated and integrated system in which predictive functionality is mostly managed by the system with help from the user (akin to existing data management functionality).

On the academic side, MauveDB [AM06] was an early system to support model-based views defined using statistical models. Such views can be used for a variety of purposes including cleaning, interpolation and prediction (with a focus on sensor network applications). The PDBMS functionality we sketch in this paper goes significantly beyond the scope of MauveDB. Our direct prediction interface and MauveDB views have similar functionality and purpose. However, we believe that automated model building and maintenance services, such as our declarative predictor interface, are essential for commoditization of predictive functionality.

Another closely related system is Fa [DB07], which was designed to support forecasting queries over time-series data. Fa offers efficient strategies for model building and selection, making a solid contribution towards model management and predictive query processing. Longview can leverage many of Fa's techniques but also aims for deeper, more comprehensive model management, by treating models as native entities and addressing the entire predictive model life cycle.

Recently, there have been successful applications of machine learning techniques to DBMS self-management problems. Query-plan-level predictions have been studied in [GKD09]. NIMO proposed techniques for accelerating the learning of cost models for scientific workflows [SBC06]. Performance prediction for concurrent query workloads was investigated in [AAB08].

6. CONCLUDING REMARKS

We argue that it is high time for the database community to start building predictive database systems. We discussed how predictive queries could meaningfully leverage and, at the same time, contribute to next generation data management. We presented our vision for a predictive DBMS called Longview, outlined the main architectural and algorithmic challenges in building it, and reported experimental results from an early case study of applying the predictive functionality for query performance prediction.

ACKNOWLEDGEMENTS

This work is supported in part by the NSF under grant IIS-0905553.

7. REFERENCES

[AAB08] Ahmad, M., Aboulnaga, A., Babu, S., and Munagala, K. Modeling and exploiting query interactions in database systems. CIKM 2008.

[ACU10] M. Akdere, U. Cetintemel, E. Upfal: Database-support for Continuous Prediction Queries over Streaming Data. PVLDB 3(1), 2010.

[AM06] A. Deshpande and S. Madden, MauveDB: Supporting Model-based User Views in Database Systems. SIGMOD 2006.

[AU07] Declarative temporal data models for sensor-driven query processing. Y. Ahmad and U. Cetintemel. DMSN 2007.

[APC08] Simultaneous Equation Systems for Query Processing on Continuous-Time Data Streams. Y. Ahmad, O. Papaemmanouil, U. Cetintemel, J. Rogers. ICDE 2008.

[BBD09] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. HotOS 2009.

[CC01] Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2006.

[DB2] DB2 Intelligent Miner Web Site. http://www01.ibm.com/software/data/iminer/

[DB07] S. Duan and S. Babu, Processing Forecasting Queries. VLDB'07.

[GH05] J. G. De Gooijer and R. J. Hyndman. 25 Years of IIF Time Series Forecasting: A Selective Review. June 2005. Tinbergen Institute Discussion Papers No. TI 05-068/4.

[GKD09] A. Ganapathi, H. Kuno, U. Dayal, J. Wiener, A. Fox, M. Jordan, D. Patterson: Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. ICDE 2009.

[GP] Google Prediction API, http://code.google.com/apis/predict/

[GZ08] Tingjian Ge, Stan Zdonik. A Skip-list Approach for Efficiently Processing Forecasting Queries. VLDB 2008.

[HR07] H. Bravo, R. Ramakrishnan. Optimizing mpf queries: decision support and probabilistic inference. SIGMOD 2007.

[IMT08] Christian Igel, Verena H., and Tobias G. Shark. *Journal of Machine Learning Research*, 2008.

[JJ08] Shantanu Joshi and Chris Jermaine. Materialized Sample Views for Database Approximation. IEEE Trans. Knowl. Data Eng. 20(3): 337-351 (2008)

[JXW08] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, P. Haas: MCDB: a monte carlo approach to managing uncertain data. SIGMOD 2008.

[MU05] Mitzenmacher, M., Upfal, E. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*.

[MWH98] Makridakis, S., Wheelwright S., and Hyndman, R. Forecasting Methods and Applications. Third Edition. John Wiley & Sons, Inc. 1998.

[Ora] Oracle Data Mining Web Site. http://www.oracle.com/technology/products/bi/odm/index.html

[SBC06] P. Shivam, S. Babu, and J. Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. VLDB 2006.

[SS08] Microsoft SQL Server 2008. www.microsoft.com/sqlserver/2008/en/us/datamining.aspx

User Feedback as a First Class Citizen in Information Integration Systems*

Khalid Belhajjame
School of Computer Science
University of Manchester
Manchester, UK
khalidb@cs.man.ac.uk

Norman W. Paton
School of Computer Science
University of Manchester
Manchester, UK
norm@cs.man.ac.uk

Alvaro A. A. Fernandes
School of Computer Science
University of Manchester
Manchester, UK
afernandes@cs.man.ac.uk

Cornelia Hedeler
School of Computer Science
University of Manchester
Manchester, UK
chedeler@cs.man.ac.uk

Suzanne M. Embury
School of Computer Science
University of Manchester
Manchester, UK
sembury@cs.man.ac.uk

ABSTRACT

User feedback is gaining momentum as a means of addressing the difficulties underlying information integration tasks. It can be used to assist users in building information integration systems and to improve the quality of existing systems, e.g., in dataspaces. Existing proposals in the area are confined to specific integration sub-problems considering a specific kind of feedback sought, in most cases, from a single user. We argue in this paper that, in order to maximize the benefits that can be drawn from user feedback, it should be considered and managed as a first class citizen. Accordingly, we present generic operations that underpin the management of feedback within information integration systems, and that are applicable to feedback of different kinds, potentially supplied by multiple users with different expectations. We present preliminary solutions that can be adopted for realizing such operations, and sketch a research agenda for the information integration community.

Keywords

User feedback, Feedback management, Information integration, Dataspaces

1. INTRODUCTION

Two decades of extensive research and real world experience suggest that building information integration systems is a difficult task [11]. Essentially, the difficulty lies in understanding user requirements as to what the relevant data sources are, and the way the contents of the sources are to be combined and structured in a form that is compatible with the user's conceptualization of the world.

To facilitate information integration, some database researchers have explored the use of feedback solicited from users. User feedback is a growing theme in information integration systems, as reflected in the number of recent proposals that utilise user feedback to guide the construction of information integration systems, and/or to improve the quality of the services they provide [1, 2, 4, 5, 13, 15, 17, 18]. For example, Talkudar *et al.* [17, 18] developed the `Q` system to assist users in creating integration queries. In doing so, the system solicits feedback on results of candidate integration queries. Feedback is also fundamental to the dataspaces vision [9], which aims to reduce the cost of setting up a data integration system, and to gradually improve the quality of the resulting system. The idea is to enlist users to provide feedback with a view to improving the quality of the services supplied by the data integration system. For example, Jeffery *et al.* [13] developed a decision-theoretic framework for specifying the order in which feedback can be solicited on a collection of schema mappings with the objective of providing the *most benefit* to a dataspace. Also, we have shown in previous work [2], that schema mappings can be selected, refined and annotated with metrics specifying their fitness to user requirements based on feedback commenting on the membership of tuples to relations in the integration schema. More recently, Doan *et al.* [7] proposed a taxonomy that classifies mass collaboration systems, namely systems that enlist a multitude of human users to help solve a given problem. In doing so, they identified issues with respect to user inputs (feedback), e.g., how to recruit and retain users, and how to evaluate their inputs.

While existing proposals showcase the key role user feedback can play within information integration systems, they are confined to the use of feedback given on a specific artifact, e.g., query [4], query result [2, 17], or mapping [1], to tackle a specific information integration sub-problem, e.g., designing a mapping [1], selecting a mapping [2], or specifying a query [18, 17]. In doing so, they do not explore the benefits that can be drawn from using feedback given on different kinds of artifacts to leverage multiple informa-

*The work reported in this paper was supported by a grant from the EPSRC.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

tion integration tasks. Moreover, they make assumptions that do not necessarily hold in practice, and that if dropped may lead to costs and losses that outweigh the benefits that can be drawn from user feedback. For example, they assume that a data integration system either has a single user (e.g., [18]), or that all the users supplying feedback have the same requirements (e.g., [13]). In practice, different users may have different expectations, in which case, the quality of the services provided by the data integration system may be seen as deteriorating in response to user feedback, at least for some users, over time, due to conflicts in requirements. Also, existing proposals assume that user requirements do not change over time. This assumption may not hold since user needs may shift, in which case previously acquired feedback may prevent the improvement of the data integration system with a view to answering new user needs.

We argue that, in order to be able to exploit different kinds of feedback provided by multiple users to leverage different information integration tasks, user feedback should be considered and managed as a first class citizen. Accordingly, we present in this paper a set of feedback management research challenges that together aim to foster the semantic cohesion of the feedback provided by users, and to increase the value and the benefits that can be drawn from acquired feedback. We propose preliminary solutions to the research challenges identified, and sketch a research agenda for the information integration community.

The paper is structured as follows. We begin by presenting a general feedback model that caters for the specification of the different kinds of feedback that have been considered in the information integration literature in Section 2. We then present operations that can be used for fostering the cohesion of the feedback provided by users by detecting inconsistencies in feedback (Section 3), and by checking the validity of feedback over time (Section 4). We go on to present operations that aim to increase the value and the benefits derived from user feedback. Specifically, we present a clustering operation for grouping users with similar expectations (in Section 5), and an operation for learning new feedback using collaborative filtering techniques (in Section 6). We close the paper in Section 7 by underlining our main contributions.

2. WHAT IS FEEDBACK?

In essence, feedback can be seen as annotations that a user provides to comment on artifacts of an information integration system, be they matches, mappings, integration schema, queries or query results, with the objective of informing the construction of the system and/or improving the quality of the services it provides. We define a feedback instance by the tuple:

$$\langle \texttt{obj}, \texttt{t}, \texttt{u}, \texttt{k} \rangle$$

specifying that the user `u` annotates the artifact `obj` using the term `t`. The user can provide different kinds of feedback. `k` indicates the kind of feedback. The set of terms that can be used depends on the kind of feedback the user wishes to provide. They can belong to controlled vocabularies or domain ontologies. In what follows, we use `uf.obj`, `uf.annot`, `uf.user` and `uf.type` to refer to `obj`, `t`, `u` and `k`, respectively.

The above feedback model is general, in that it can be used to describe the different kinds of feedback defined in the information integration literature. Table 1 presents prominent proposals in the field that use feedback, and shows how the feedback used in each proposal can be described using the model presented above.

We present in what follows some examples of feedback defined in the literature and show how they can be mapped to the above model.

Example 2.1. Consider the `Q` system developed by Talkudar *et al.* [18]. This system assists users in creating integration queries that span multiple data sources. To learn user preference as to which query captures expectations, the user supplies feedback by ordering result tuples returned by different alternative queries. For example, consider two tuples $\texttt{t}_1$ and $\texttt{t}_2$ retrieved by the alternative queries $\texttt{q}_1$ and $\texttt{q}_2$, respectively. If the user judges that $\texttt{t}_1$ meets the expectations better than $\texttt{t}_2$, then $\texttt{t}_1$ is ranked before $\texttt{t}_2$. Using our model, such feedback can be specified as follows: $\langle\langle \texttt{t}_1, \texttt{t}_2 \rangle, \texttt{before}, \texttt{u}, \texttt{ranking} \rangle$. The vocabulary used for annotation, in this case, is composed of two terms `before` and `after`.

Example 2.2. McCann *et al.* [15] developed a system that informs schema matching by soliciting feedback from users. As an example, consider a binary match $\langle \texttt{r}_1, \texttt{r}_2 \rangle$ that associates the relations $\texttt{r}_1$ and $\texttt{r}_2$, and consider that the user `Anhai` specified that such a match is incorrect. Using our model, such feedback can be specified as follows: $\langle\langle \texttt{r}_1, \texttt{r}_2 \rangle, \texttt{fp}, \texttt{Anhai}, \texttt{match\_correctness} \rangle$. Using the same system, users can provide feedback that specifies (or confirms) the data type of a given attribute. For example, the same user can confirm that the attribute `fecha` is of type `date`, which can be described using our feedback model as follows: $\langle \texttt{fecha}, \texttt{date}, \texttt{Anhai}, \texttt{type\_correctness} \rangle$.

Example 2.3. We showed in a previous work [2] that feedback can be used to annotate schema mappings with estimates specifying the degree to which they meet user expectations. In doing so, a user annotates result tuples that are retrieved using candidate mappings to populate a given relation in the integration schema as true positives, false positives or false negatives. Consider a relation `r` in the integration schema, a mapping `m` used to populate `r` using data from the sources, and a tuple `t` of `r` that is retrieved using `m`. The following feedback instance, $\langle\langle \texttt{r}, \texttt{m}, \texttt{t} \rangle, \texttt{tp}, \texttt{Norman}, \texttt{tuple\_membership} \rangle$, specifies that `t` is a true positive, i.e., that `t` is a member of `r` according to the expectations of the user `Norman`. Using feedback instances of the above form, candidate mappings for populating the elements of an integration schema can be annotated, selected and refined [2].

Having illustrated how the model presented in this section can be used to describe different kinds of feedback, we now present operations that act on feedback instances of that model with the goal of determining the validity of the feedback instances provided by users, and to increase the benefits that can be derived from their use.

3. FEEDBACK INCONSISTENCY

Different users may have different requirements. Moreover, the requirements of the same user may change over time.

Table 1: Kinds of feedback considered in the information integration literature.

Proposal	Objects on which feedback is given	Set of terms used for annotating objects
Alexe *et al.* [1]	an instance of a given schema and the instance obtained by its transformation into another schema	{'yes', 'no'} // used to comment on schema transformation
Belhajjame *et al.* [2]	a result tuple	{'true positive', 'false positive', 'false negative'}
	an attribute and its value	{'true positive', 'false positive', 'false negative'}
Cao *et al.* [4]	a candidate query	{'true positive', 'false positive'}
	a pair of candidate queries	{'before', 'after'} // used for ordering queries
Chai *et al.* [5]	a view result tuple	{'insert', 'delete', 'update'}
Jeffery *et al.* [13]	a mapping	{'true positive', 'false positive'}
McCann *et al.* [15]	a relation attribute	set of attribute data types
	two attributes of a given relation	set of constraints
	a match	{'true positive', 'false positive'}
Talkudar *et al.* [18]	a result tuple	{'true positive', 'false positive'}
	a pair of result tuples	{'before', 'after'} // used for ordering results

In this section, we present an approach to detecting these phenomena, viz. feedback inconsistency. The basic idea is that if two users have different requirements, then this difference may give rise to conflicts between the feedback instances they supply. Similarly, if the requirements of a given user changes over time, then this change may give rise to inconsistency between feedback instances supplied at different points in time. Given two feedback instances $\texttt{uf}_1$ and $\texttt{uf}_2$, we present rules that can be used to check whether $\texttt{uf}_1$ and $\texttt{uf}_2$ are inconsistent.

We start by considering the case in which $\texttt{uf}_1$ and $\texttt{uf}_2$ annotate the same object. $\texttt{uf}_1$ and $\texttt{uf}_2$ are inconsistent if they are of the same kind, and annotate the same object using conflicting terms. Below is a rule that detects this type of inconsistency.

```
1 inconsistent(uf1, uf2) : −
2   - uf1 and uf2 are of the same kind
3   uf1.type = uf2.type,
4   - uf1 and uf2 label the same object
5   uf1.obj = uf2.obj,
6   - using inconsistent annotation
7    uf1.type.conflictAnnotations(uf1.annot, uf2.annot)
```

Notice that the notion of inconsistency depends on the kinds of feedback *(line 7)*; in this respect, the function `conflictAnnotations()` acts as an extensibility point. As a proof of concept, we present below two definitions of this function for two different kinds of feedback.

Example 3.1. Consider for example that feedback annotates query tuples as true positives or false positives (e.g., [2]). In this case, two terms are conflicting if they are different. That is:

```
1 conflictAnnotations(c1, c2) : −
2   - The terms c1 and c2 are different
3    c1 ≠ c2
```

Example 3.2. Consider now that the feedback provided by users uses terms taken from domain ontologies (e.g., [15]). In this case, two concepts are conflicting if they are disjoint. That is:

```
1 conflictAnnotations(c1, c2) : −
2   - The data types denoted by c1 and c2
3   - are disjoint
4    c1 ⊓ c2 = ⊥
```

The artifacts that constitute an information integration system can be dependent on each other. Such dependencies may give rise to constraints between feedback, more specifically between the terms used to annotate dependent artifacts. If such constraints are not satisfied, then this results in inconsistency between the feedback instances in question. This form of inconsistency can be detected using the following rule:

```
1 inconsistent(uf1, uf2) : −
2   - uf1 and uf2 are of the same kind
3   uf1.type = uf2.type,
4   - there is a pair of dependency and constraint
5   - ⟨dep_type, const_type⟩ that are associated with
6   - the kind of feedback uf1.type
7   ∃ ⟨dep_type, const_type⟩ ∈ uf1.type.getDepConstPair(),
8   - the objects annotated by uf1 and uf2 are
9   - related using the dep_type dependency
10  dependent(uf1.obj, uf2.obj, dep_type),
11  - the annotations assigned by the uf1 and uf2
12  - do not satisfy the const_type constraint
13  ¬constraint(uf1.annot, uf2.annot, const_type)
```

where getDepConstPair() is a function that return pairs of dependency type and constraint type that are associated with a given feedback type, $\texttt{dependent}(\texttt{uf}_1.\texttt{obj}, \texttt{uf}_2.\texttt{obj}, \texttt{dep}_{\texttt{type}})$ is a predicate that is true if the objects $\texttt{uf}_1.\texttt{obj}$ and $\texttt{uf}_2.\texttt{obj}$ are related by a dependency of type $\texttt{dep}_{\texttt{type}}$, and $\texttt{constraint}(\texttt{uf}_1.\texttt{annot}, \texttt{uf}_2.\texttt{annot}, \texttt{const}_{\texttt{type}})$ is a predicate that is true if the constraint of type $\texttt{const}_{\texttt{type}}$ holds between the annotations $\texttt{uf}_1.\texttt{annot}$ and $\texttt{uf}_2.\texttt{annot}$.

Example 3.3. As an example, consider the following feedback instances which annotate values of relation attributes as true positives or false positives:

- $\texttt{uf} = \langle\langle \texttt{r}.\texttt{att}, \texttt{v}\rangle, \texttt{tp}, \texttt{Norman}, \texttt{value\_membership}\rangle$ specifies that the value v of the attribute att of the r relation is a true positive, and
- $\texttt{uf}' = \langle\langle \texttt{r}'.\texttt{att}', \texttt{v}\rangle, \texttt{fp}, \texttt{Norman}, \texttt{value\_membership}\rangle$ specifies that the value v of the attribute $\texttt{att}'$ of the $\texttt{r}'$ relation is a false positive.

Additionally, consider that there is a foreign key that links the attribute att of r to the attribute $\texttt{att}'$ of $\texttt{r}'$. Given this, we can deduce that uf and $\texttt{uf}'$ are inconsistent. Indeed, if v is a true positive for the attribute att of r, then it should also be a true positive for the attribute $\texttt{att}'$ of $\texttt{r}'$ given the inclusion constraint implied by the foreign key. The predicates dependent and constraint used for detecting inconsistencies of the above form can be defined as follows:

- $\texttt{dependent}(\texttt{uf}_1.\texttt{obj}, \texttt{uf}_2.\texttt{obj}, \text{`}\texttt{foreign\_key}'\text{)}$ is true if the attribute in $\texttt{uf}_1.\texttt{obj}$ is linked to the attribute in $\texttt{uf}_2.\texttt{obj}$ using a foreign key, and the attribute values in $\texttt{uf}_1.\texttt{obj}$ and $\texttt{uf}_2.\texttt{obj}$ are the same, and is false otherwise.
- $\texttt{constraint}(\texttt{uf}_1.\texttt{annot}, \texttt{uf}_2.\texttt{annot}, \text{`}\texttt{tp\_inclusion}'\text{)}$ is false if $\texttt{uf}_1.\texttt{annot}$ takes the value tp and $\texttt{uf}_2.\texttt{annot}$ takes the value fp, and is true otherwise. That is:

```
1 constraint(annot1, annot1, 'tp_inclusion') : −
2   ¬((annot1 = tp), (annot2 = fp)),
```

4. FEEDBACK VALIDITY

User requirements may change over time in which case previously acquired feedback may become invalid. In this section, we specify the situations under which a feedback instance can be stated to be valid or invalid.

We use the event calculus [14] as a formalism to define the rules used to check the validity of user feedback. The event calculus is a logic-based formalism that can be used to deduce the state of a given domain based on the events (actions) that have taken place. In the case of our analysis, the main event is the fact that a user supplies a feedback instance uf. We denote this event by supplyFeedback(uf). In addition we consider the following predicates:

- exists(obj) is true if the object obj exists, and is false, otherwise.
- valid(uf) is true if the feedback instance uf is valid, and is false, otherwise.
- invalid(uf) is true if the feedback instance uf is invalid, and is false, otherwise.
- hasUnkownStatus(uf) is true if the status of uf is unknown, and is false if uf is either valid or invalid.

Notice that the above predicates, exists(obj), valid(uf), invalid(uf) and hasUnkownStatus(uf), are fluent: their value is subject to change over time. As well as the above predicates, we consider two predicates that are specific to the event calculus, viz. holdsAt and happens. holdsAt(fl, t) and happens(e, t) are used to check that a given fluent fl holds at a given time point t and to check that a given event e occurs at a given time point t, respectively. We will now specify the cases in which a feedback instance is valid, invalid or has unknown status.

We start by specifying the conditions under which a feedback instance $\texttt{uf}_1$ is invalid. $\texttt{uf}_1$ is invalid if there is a valid feedback instance $\texttt{uf}_2$ that is conflicting with and fresher than $\texttt{uf}_1$. This is because we consider that inconsistencies in feedback emerge from changes in requirements, in which case, $\texttt{uf}_1$ reflects past requirements that have changed, and as a result, is no longer valid. We formally define the rule for identifying invalid feedback instances as follows:

```
1 holdsAt(invalid(uf1), t) : −
2   - uf1 was supplied by the user at a time point
3   - that is equal to or before t
4   happens(supplyFeedback(uf1), t1), t1 ≤ t,
5   - There exists a feedback instance uf2 that was
6   - supplied by the user at a time point that
7   - is equal to or less than t, and greater than t1
8   happens(supplyFeedback(uf2), t2), t1 < t2 ≤ t,
9   - uf2 is inconsistent with uf1,
10  - and is known to be valid at t
11  inconsistent(uf1, uf2), holdsAt(valid(uf2), t)
```

We now specify the conditions under which a feedback instance $\texttt{uf}_1$ is valid. $\texttt{uf}_1$ is valid at the time point t if it was supplied by the user before t, there is no conflicting feedback instance $\texttt{uf}_2$ that is fresher than $\texttt{uf}_1$ and valid at t, and the object $\texttt{uf}_1.\texttt{obj}$ on which feedback is given exists at t. We do not consider feedback that was given on an object that is no longer available. This is because the non existence of the object in question may in certain cases mean that the annotation given by the feedback is no longer valid. To illustrate this, consider the following relational table car(model, manufacturer), and consider that the user gave a feedback instance uf annotating the tuple $\langle\texttt{Mini}, \texttt{Rover}\rangle$ as a true positive of the car relation. The Mini manufacturer changed later on, and the tuple was updated to $\texttt{car} = \langle\texttt{Mini}, \texttt{BMW}\rangle$. As a result of this update, the feedback instance uf given previously is no longer valid. We formally define the rule for identifying valid feedback instances as follows:

```
1 holdsAt(valid(uf1), t) : −
```

```
2   - uf1 was supplied by the user at a time point
3   - that is equal to or before t
4   happens(supplyFeedback(uf1), t1), t1 ≤ t,
5   - There is no valid feedback instance uf2 that
6   - is conflicting with and fresher than uf1
7   not (happens(supplyFeedback(uf2), t2),
8        inconsistent(uf1, uf2),
9        holdsAt(valid(uf2), t)), t1 < t2 ≤ t),
10  - The object that uf1 annotates exists at t
11  holdsAt(exists(uf1.obj), t).
```

Notice that the above rule is recursive. Specifically, the validity of the feedback instance $\texttt{uf}_1$ is preconditioned by the non existence of a valid feedback instance $\texttt{uf}_2$ that is conflicting with and fresher than $\texttt{uf}_1$.

There are situations in which we are unable to determine whether a feedback instance is valid or invalid. This is the case, for example, when the object on which feedback was given, is no longer available. This is also the case if the user supplies two inconsistent feedback instances at the same time. (The latter case may be indicative of malicious behaviour of the user [15].) The status of a feedback instance $\texttt{uf}_1$ is unknown if it is neither valid nor invalid. That is:

```
1 holdsAt(hasUnkownStatus(uf1), t) : −
2   - uf1 was supplied by the user before t
3   happens(supplyFeedback(uf1), t1), t1 ≤ t,
4   - uf1 is not known to be valid at t
5   not holdsAt(valid(uf1), t),
6   - uf1 is not known to be invalid at t
7   not holdsAt(invalid(uf1), t).
```

Note that the above analysis on validity of user feedback can be applied to feedback instances that are supplied by the same user, or by a group of users that are known to have the same requirements. This raises the question as to how to determine whether a group of users have the same requirements. In the following section, we show how clustering techniques can be used for this purpose.

5. CLUSTERING USERS BASED ON FEEDBACK

The users of an information integration system should have the same (or similar) requirements to ensure the cohesion of the feedback they provide. Checking the consistency of the feedback provided by different users can be used as a means for ensuring such cohesion. In this section, we explore another technique that can be used for this purpose, namely clustering.

Clustering is unsupervised classification [12]. We use this technique to group users according to their requirements. Specifically, given a set of users and their (partial) requirements, elicited through feedback, clustering is used to identify groups of users with similar requirements. Clustering presents the following potential benefits:

- The feedback provided by the users within a cluster can be used collectively, thereby improving the quality of the integration system.
- Clustering may reveal that the users of an existing information system have different requirements, and therefore point out the need to create multiple information integration systems to replace the existing one.
- Conversely, clustering may identify opportunities for grouping the users of two or more information integration systems into a single group, if it turns out that they have similar requirements.
- Clustering can also be used to identify outlier users within an information integration system, i.e., users that have different requirements from the rest of the users.

As a proof of concept, we present in what follows an example illustrating how users can be clustered according to their requirements.

Example 5.1. Consider a set of users who would like to have information about available proteins. Specifically, there is a `Protein` relation in the integration schema that users wish to query. Not all users have the same requirements, e.g., they may be interested in proteins belonging to different species or with different structures, motifs, etc. Clustering can be used to identify groups of users with similar requirements as to which tuples should populate the `Protein` relation. There are many algorithms for clustering that are readily available in the literature [12]. To make use of such algorithms, however, we need to choose a pattern representation for specifying user requirements, and a metric for measuring the distance between user requirements. In what follows, we present a pattern representation and a distance metric that are suitable for the above clustering problem.

Pattern Representation. We specify a user's requirements with respect to the extent of the `Protein` relation using a pair $\langle \texttt{E}, \texttt{UE} \rangle$, where $\texttt{E}$ is the set of tuples that are expected by the user, i.e., the tuples that the user annotated as true positives or false negatives, and $\texttt{UE}$ is the set of tuples that are not expected by the user, i.e., the tuples that the user annotated as false positives.

Distance metric. To define the distance between requirements, we start by defining the notion of similarity of requirements. Consider two users $\texttt{u}_1$ and $\texttt{u}_2$. The larger the overlap between the expected tuples of $\texttt{u}_1$ and $\texttt{u}_2$, i.e., $\texttt{E}_1$ and $\texttt{E}_2$, and the larger the overlap between their unexpected tuples, $\texttt{UE}_1$ and $\texttt{UE}_2$, the more similar are their requirements are. A similarity measure that captures this property can be defined as follows:

$$\texttt{sim}(\texttt{u}_1, \texttt{u}_2) = \texttt{w}_\texttt{e} \times \frac{|\texttt{E}_1 \cap \texttt{E}_2|}{|\texttt{E}_1 \cup \texttt{E}_2|} + \texttt{w}_\texttt{ue} \times \frac{|\texttt{UE}_1 \cap \texttt{UE}_2|}{|\texttt{UE}_1 \cup \texttt{UE}_2|}$$

where $\texttt{w}_\texttt{e}$ and $\texttt{w}_\texttt{ue}$ are weights such that $\texttt{w}_\texttt{e} + \texttt{w}_\texttt{ue} = 1$. The ratios $\frac{|E_1 \cap E_2|}{|E_1 \cup E_2|}$ and $\frac{|UE_1 \cap UE_2|}{|UE_1 \cup UE_2|}$ are known as the Jaccard

coefficient [3]. The above similarity measure takes values between 0 and 1, and the corresponding distance metric can be defined as follows: $\mathtt{dis}(u_1, u_2) = 1 - \mathtt{sim}(u_1, u_2)$.

Using the representation and distance function defined above, we can group users of the Protein relation using existing clustering algorithms [3]. In what follows, we present preliminary empirical results with synthetic data show how such users can be clustered into groups using hierarchical agglomerative clustering. Specifically, we consider the following setting. A set of 20 users, each of which provides feedback instances identifying expected and unexpected tuples of the Protein relation. Regarding the distance metric, we consider that a feedback instance specifying an expected tuple has the same weight as a feedback instance specifying an unexpected tuple, i.e., $w_e = w_{ue} = 0.5$. To cluster users, we use the *hclust* algorithm provided by the R statistical framework[1] for performing hierarchical clustering. Specifically, we run the following procedure:

```
1 For each user u_i
2    Generate a set of feedback instances UF_{u_i}
3 For each pair of users ⟨u_i, u_j⟩
4    Compute the distance between them
5    Log that distance in the distance matrix DM
6 Invoke the hclust program using as input DM
```

DM is a two-dimensional array that is used to log the distances, taken pairwise, of a set of users.

We run the above procedure by varying the size of overlap in feedback instances between users. Specifically, we considered the following three scenarios:

1. The overlap in feedback instances between users is small: less than 5 percent of the feedback instances supplied by a given user overlap with the feedback instances supplied by another user.
2. The overlap in feedback instances between users is significant: 10 to 20 percent of the feedback instances that are provided by a given user overlap with the feedback instances supplied by another user.
3. The overlap in feedback instances between users is large: 20 to 50 percent of the feedback instances that are provided by a given user overlap with the feedback instances supplied by another user.

Figures 1, 2 and 3 visualize the clustering results obtained in each of the above scenarios using dendrograms. The users, which are identified by integers, are listed along the x-axis in an order that is convenient for showing the cluster structure. The y-axis measures inter-cluster distance. Consider, for example, the users identified by the integers 13 and 17 in Figure 3. They are joined into the same cluster. The distance between them is 0.1, and no other users are separated by a distance smaller than that value. Consider Figures 1, 2 and 3 together. As expected, the comparison of three dendrograms shows that the smaller the overlap between users in terms of feedback, the poorer the quality of the clustering. Indeed, the intra-cluster distance is large when overlap in feedback is small (Figure 1). A cluster, therefore, may contain users with requirements that are significantly different. On the other hand, the quality of clustering is better when overlap in feedback is large. The intra-cluster distance is reduced (Figure 3), thereby yielding clusters that join users with similar requirements.

[1] http://www.r-project.org

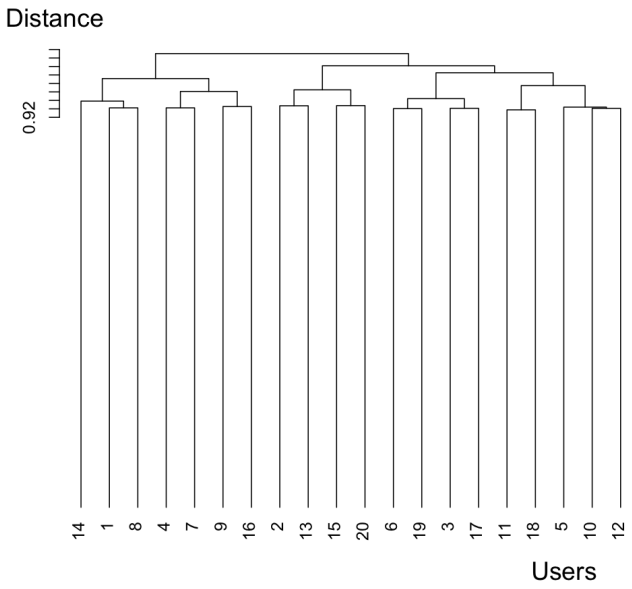


Figure 1: Clustering results when overlap in user feedback is small

The above example considers feedback instances of the same kind. In practice, different kinds of feedback instances given on different kinds of artifacts will be used to elicit user requirements. We can use feedback propagation as a mechanism to deal with the issue. The basic idea can be formulated as follows: given a feedback instance uf annotating an object of a given type, e.g., a mapping m, derive a feedback instance $\mathtt{uf}'$ annotating an object of another type, e.g., a tuple t that is produced using the mapping m. Propagating feedback may in certain situations be used for aggregating feedback instances of different types.

Aggregation is also another means that can be explored when feedback instances are of different kinds. As an example, assume that we want to cluster users based on feedback instances of different kinds. Using the approach described above, we obtain a set of clusterings $\{C_1, \ldots, C_n\}$; C_i being the clustering obtained using feedback instances of the ith type. The obtained clusterings can then be aggregated to produce a single clustering C that agrees as much as possible with the n clusterings [10]. We can do so, for example, by creating a graph G, the vertices of which represent users, and where the edges are weighted using the information provided by the clusterings $C_1, \ldots, C_n$. Specifically, the weight of the edge $\langle u_1, u_2 \rangle$ linking the users u_1 and u_2 is the fraction

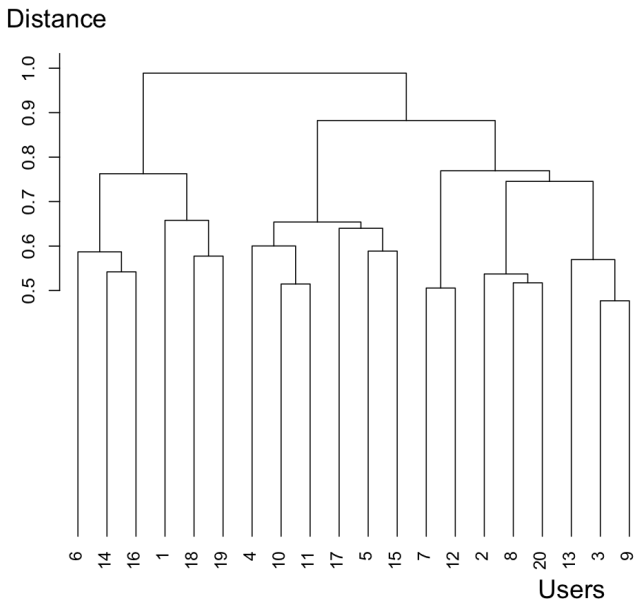


Figure 2: Clustering results when overlap in user feedback is important

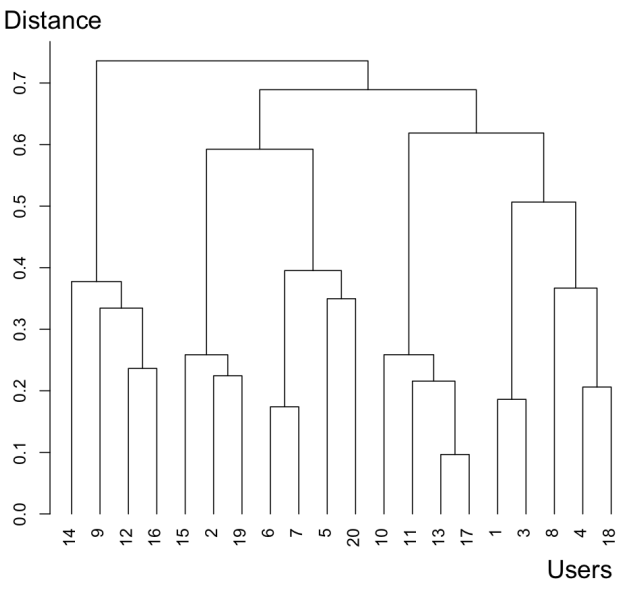


Figure 3: Clustering results when overlap in user feedback is large

of clusterings in which the two users are placed in different clusters. The cluster `C` is then obtained using a partitioning of the graph `G` that cuts as few as possible of the edges with small weight, e.g., less than `0.5`, and cuts as many as possible of the edges with large weight, e.g., more than `0.5`. Multidimensional clustering algotithms, e.g., PCA, is another means that can be explored for grouping users based on feedback instances of different types [3], e.g., each dimension can be used to encode the user requirements that are derived from feedback instances of a given kind.

As mentioned at the beginning of this section, clustering can be used as a means for reducing the workload of individual users in terms of the feedback they provide. We present in the following section a technique that can be used for automatically generating user feedback.

6. LEARNING FEEDBACK USING COLLABORATIVE FILTERING

There are situations in which it is desirable to learn the requirements of a given user. For example, a user may have only partial knowledge of requirements, in which case, s/he may not be in a position to provide feedback on given artifacts. Also, if a new user joins an information integration system, then we may want to accelerate the requirement acquisition process. In this section, we explore the use of collaborative filtering [16], a technology that is used for learning user preferences, to learn user feedback. Specifically, the problem that we tackle can be formulated as follows. Given a user `u` and an object `obj` of an information integration system, predict the feedback `uf` that `u` would provide to annotate `obj` based on other users' feedback. The underlying assumption to our approach is that if two users have similar requirements, then they are likely to provide the same feedback on a given artifact.

Example 6.1. Assume, for example, that we would like to know whether a given tuple `t` can be used to populate a relation `r` of the integration schema in the conceptualization of the user `u`. We can learn the feedback the user `u` would provide to annotate `t` by using the feedback instances that other users provided to annotate that tuple. Assume that the users $\mathtt{u}_1, \ldots, \mathtt{u}_n$ provided feedback instances annotating `t`. The following formula computes a score, $\mathtt{expected}(\mathtt{t}, \mathtt{u})$, estimating the likelihood that `t` is expected and a score, $\mathtt{unexpected}(\mathtt{t}, \mathtt{u})$, estimating the likelihood that `t` is unexpected for `u`.

$$\mathtt{expected}(\mathtt{t},\mathtt{u}) = \frac{\sum_{\mathtt{i}=1}^{\mathtt{i}=\mathtt{n}}(\mathtt{sim}(\mathtt{u},\mathtt{u_i}) \times \mathtt{isExp}(\mathtt{u_i},\mathtt{t}))}{n}$$

$$\mathtt{unexpected}(\mathtt{t},\mathtt{u}) = \frac{\sum_{\mathtt{i}=1}^{\mathtt{i}=\mathtt{n}}(\mathtt{sim}(\mathtt{u},\mathtt{u_i}) \times \mathtt{isUnexp}(\mathtt{u_i},\mathtt{t}))}{n}$$

$\mathtt{expected}(\mathtt{t},\mathtt{u})$ and $\mathtt{unexpected}(\mathtt{t},\mathtt{u})$ are calculated as weighted averages of the feedback given by other users where the weights are the similarity scores between user requirements. `sim`() is the similarity measure defined in the previous section. $\mathtt{isExp}(\mathtt{u_i},\mathtt{t})$ (resp. $\mathtt{isUnexp}(\mathtt{u_i},\mathtt{t})$) is a boolean predicate that is true if the tuple `t` is expected (resp. unexpected) in the conceptualization of the user $\mathtt{u_i}$.

The operations that we presented for detecting inconsistencies in feedback, clustering users and learning feedback operate on feedback given on common objects. Therefore, the results they produce may not be trustworthy when the set of objects annotated by more than one feedback instance is small. To improve the quality of the results produced by the operations presented in this paper, we can take inspiration from model-based collaborative filtering algorithms [6]. Rather than making predictions based on items that are

commonly annotated by users, model-based collaborative filtering algorithms employ user annotations to derive, for each user, a model characterizing his/her preferences. Predictions are then made by matching those models. We can adopt a similar approach. The issue here is, of course, which model to adopt for synthesizing user requirements elicited through feedback.

As an example, in our previous work [2], users provide feedback commenting on the membership of tuples to a relation in the integration schema. If the set of tuples annotated by more than one user is small then the operation we presented for learning feedback may not be able to generate meaningful feedback.

Feedback that comments on tuple membership was used in [2] to annotate schema mappings with metrics estimating their precision and recall. Specifically, given $\mathtt{R}$, a relational table, and $\mathtt{m}$, a mapping that is a candidate to populate $\mathtt{R}$, the precision of $\mathtt{m}$ given the feedback $\mathtt{UF}_{\mathtt{u_i}}$ supplied by the user $\mathtt{u_i}$ is defined as the ratio of the number of true positive tuples returned by $\mathtt{m}$ to the sum of the number of true positive tuples and the number of false positive tuples of $\mathtt{m}$ given the feedback instances in $\mathtt{UF}_{\mathtt{u_i}}$. Similarly, the recall of $\mathtt{m}$ is defined as the ratio of the number of true positive tuples returned by $\mathtt{m}$ to the sum of the number of true positive tuples and the number of false negative tuples of $\mathtt{m}$ given $\mathtt{UF}_{\mathtt{u_i}}$. That is:

$$\mathtt{Precision}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}}) = \frac{|\mathtt{tp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})|}{|\mathtt{tp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})| + |\mathtt{fp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})|}$$

$$\mathtt{Recall}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}}) = \frac{|\mathtt{tp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})|}{|\mathtt{tp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})| + |\mathtt{fn}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})|}$$

where $|\mathtt{s}|$ denotes the magnitude of the set s, and $\mathtt{tp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})$, $\mathtt{fp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})$ and $\mathtt{tp}(\mathtt{m}, \mathtt{UF}_{\mathtt{u_i}})$ denote, respectively, the sets of true positives, false positives and false negatives of $\mathtt{m}$ given the feedback instances in $\mathtt{UF}_{\mathtt{u_i}}$.

When overlaps between the sets of feedback instances provided by different users are small, the approach we described for learning feedback is likely to be ineffective: predictions will be made using little or no evidence, and as such are likely to be inaccurate. To overcome this problem, we can make use of the precision and recall estimates defined above to learn feedback. Indeed, such estimates synthesize user requirements as to the quality of the mapping used to populate a given relation. Therefore, they can be used to compare users' requirements. Users with similar requirements are likely to be associated with close estimates, and, conversely, users with different requirements are likely to be associated with disparate estimates. Using such estimates, we can therefore learn the feedback instance a given user $\mathtt{u_i}$ would provide to annotate a given tuple $\mathtt{t}$ by matching the precision and recall estimates computed based on the feedback supplied by $\mathtt{u_i}$ with the precision and recall estimates computed for other users.

As well as collaborative learning techniques, feedback can be inferred by exploiting available domain knowledge. To illustrate this, consider the two following relational tables $\mathtt{Protein}(\mathtt{name}, \mathtt{accession}, \mathtt{gene}, \mathtt{pfam})$ and $\mathtt{ProteinFamily}(\mathtt{id}, \mathtt{desc})$ and consider a referential integrity constraint specifying that a value of the attribute `pfam` of the `Protein` relation is also a value of the attribute `id` of the `ProteinFamily` relation. Consider now that the user supplies a feedback instance specifying that `PF05387` is an expected value of the `pfam` attribute. Given the above integrity constraint, we can derive a feedback instance specifying that the value `PF05387` is an expected value of the `id` attribute. Likewise, if the user specifies that a given instance is an unexpected value for the `id` attribute, then we can derive a feedback instance specifying that such a value is unexpected for the `pfam` attribute.

It is also worth mentioning that feedback provides partial information about user requirements. We cannot expect users to provide feedback on every artifact. Instead, in most cases, users will be willing to provide feedback on a small subset of the objects they are presented with. Because of the scarcity of the feedback, the results of the operations presented for verifying the validity of feedback, clustering users and learning feedback may be wrong or misleading, e.g., an invalid feedback instance may be found to be valid, or two users with different requirements can be grouped into the same cluster. This raises the question as to how to manage uncertainty in the results produced by the operations given the feedback used as input. A possible solution that can be explored consists in devising metrics, such as those adopted in [8, 2], to estimate the degree to which the results of an operation are correct based on the proportion of objects that are annotated by feedback.

7. CONCLUSIONS

While it is recognised that user feedback can play a central role in dealing with the difficulties underlying the construction and maintenance of information integration systems, existing proposals, e.g., [2, 17, 4, 11], address specific integration sub-problems considering a specific kind of feedback sought, in most cases from a single user, to annotate a specific kind of artifacts. We have shown in this paper that treating feedback as a first class citizen presents several advantages. Specifically, we presented a straightforward model for describing different kinds of feedback that have been considered in the information integration literature. We proposed operations that can underpin the management of feedback within information integration systems, and that are applicable to feedback instances of the model we proposed. We illustrated through examples how such operations can be used to maximize the benefits that can be derived from feedback compared with existing proposals. In particular, the operations presented can be applied to feedback instances of different kinds supplied by users with different and changing requirements. We have presented preliminary solutions that can be adopted in the realization of such operations. Furthermore, we identified issues (and therefore research opportunities) that underlie feedback management in information integration systems.

8. REFERENCES

[1] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 10–19. IEEE, 2008.

[2] K. Belhajjame, N. W. Paton, S. M. Embury, A. A. A. Fernandes, and C. Hedeler. Feedback-based annotation, selection and refinement of schema mappings for dataspaces. In *Proceedings of the 13th International Conference on Extending Database Technology, EDBT 2010, Lausanne, Switzerland*, pages 573–584. ACM, 2010.

[3] P. Berkhin. A survey of clustering data mining techniques. In J. Kogan, C. Nicholas, and M. Teboulle, editors, *Grouping Multidimensional Data: Recent Advances in Clustering*, pages 25–71. Springer, 2006.

[4] H. Cao, Y. Qi, K. S. Candan, and M. L. Sapino. Feedback-driven result ranking and query refinement for exploring semi-structured data collections. In *Proceedings of the 13th International Conference on Extending Database Technology, EDBT 2010, Lausanne, Switzerland*, pages 3–14. ACM, 2010.

[5] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA*, pages 87–100. ACM, 2009.

[6] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada*, pages 271–280. ACM, 2007.

[7] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Mass collaboration systems on the world-wide web. *Commun. ACM*. To appear.

[8] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. *VLDB J.*, 18(2):469–500, 2009.

[9] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.

[10] A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation. *TKDD*, 1(1), 2007.

[11] A. Y. Halevy, A. Rajaraman, and J. J. Ordille. Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea*, pages 9–16. ACM, 2006.

[12] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[13] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada*.

[14] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.

[15] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, Cancún, México*, pages 110–119. IEEE, 2008.

[16] X. Su and T. M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009:2–2, 2009.

[17] P. P. Talukdar, Z. G. Ives, and F. Pereira. Automatically incorporating new sources in keyword search-based data integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA*.

[18] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *PVLDB*, 1(1):785–796, 2008.

Data Externality

Rakesh Agrawal
Search Labs, Microsoft Research
1065 La Avenida
Mountain View, CA 94043
rakeshA@microsoft.com

ABSTRACT

In economics, an externality is an indirect effect of consumption or production activity on agents other than the originator of such activity. We observe that internet is enabling the design of information services that become smarter more they are used because of the data generated in the process. We give examples from web search to make the notion of data externality concrete and propose that thinking and designing for data externality could be an interesting direction for future data research.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications – *Data Mining*. H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Search Process*. H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Web-based services.*

General Terms

Algorithms, Design, Economics, Human Factors

Keywords

Externality, Web Search, Health, Education

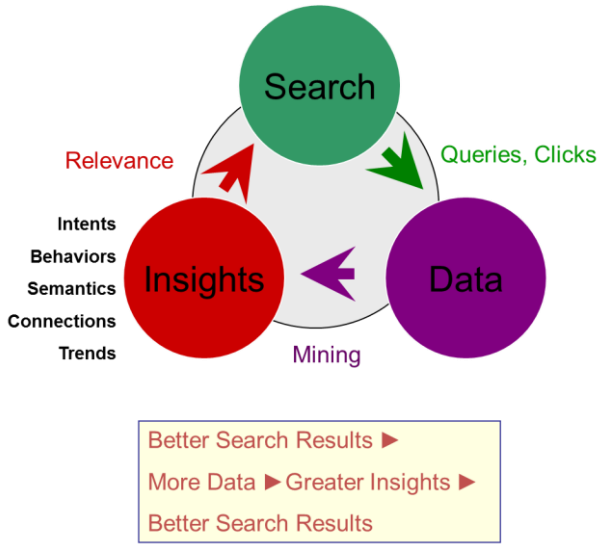
1. INTRODUCTION

Externalities are indirect effects of consumption or production activity, that is, effects on agents other than the originator of such activity which do not work through the price system [1]. A well-known example of positive externality is the network effect - an individual buying a device that is interconnected in a network increases the usefulness of similar devices to other people who already have them without having to pay extra for this increase in their utility function.

Internet is enabling the design of information services that become increasingly smarter more they are used by making use of the data which is generated as the users interact with the service. We call this positive spillover effect the *data externality*. However, every online information service does not automatically benefit from this effect. They need to be carefully engineered for them to exhibit positive data externality. Our view is that the data researchers can play pivotal role in the design and deployment of such information services.

In order to make the discussion concrete, we first discuss data externality in web search. Specifically, we describe two representative applications that have been designed with data externality in mind. We then consider two other areas – health and education – and suggest how data externality can be incorporated in these domains. We conclude by outlining some research opportunities that data externality offers to us data researchers.

2. SEARCH & DATA: VIRTUOUS CYCLE



There exists a strong virtuous cycle between web search and user data. As users pose search queries and click on search results, these queries and clicks are recorded by the search engine, which are then mined by the search engine to develop greater insights, which in turn help search engines provide better search results, leading to increased number of queries and clicks at the search engine. The greater the number of queries and clicks a search engines sees, the better it can get in producing more relevant results. We describe next two illustrative instances of this phenomenon.

2.1 Ranking

A key determinant of user satisfaction with search results is the quality of ranking. A popular way of ranking web results is by employing a learning algorithm [2]. The performance of a learning algorithm critically depends on the quantity and quality of training data [3]. The training data for this purpose consists of

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) , January 9-12, 2011, Asilomar, California, USA.

tuples of the form <*q, d, l*> where the label *l* is the relevance judgment assigned to the document *d* for a query *q*. The judgment can be on a multi-point scale, ranging from perfect to bad. For each training tuple, certain number of query independent features (e.g. static page rank of the document) as well as query dependent features (e.g. position of a query word in the title of the document) are computed, which are then used for training the learning algorithm. But this begs the question – how are the <*q, d, l*> tuples generated in the first place?

Queries used in the training data are sampled from the query log. Clearly, the larger the number of queries a search engine sees, the more representative will be the sample. Trickier is the problem of obtaining labels for query-document pairs. Labels can be generated by employing human judges. However, obtaining a large number of judgments, particularly when one is dealing with documents in multiple languages can be expensive and time consuming. A more subtle issue is that the task of manual judging is not easy. Consider the query, *wink*, and two documents: one that is the home page of the Winks statistical software and another that contains various emoticons. How would you judge them, when you do not know what was the intent of the query?

The way out of this dilemma posited in [4] is to use the log of page impressions and documents clicked by various users for a given query and use this data for algorithmically generating labels. The exact details of the algorithm are not important for the purposes of this paper. The essential idea is to construct a per-query preference graph in which documents constitute vertices and weights on edges between vertices correspond to estimated number of users who prefer one document over the other based on click frequencies. Labels are then assigned to vertices by cutting the graph into as many partitions as the desired number of labels in such a way that the difference in the weights of edges that agree with the labeling and those that disagree is maximized, and thus maximizing user satisfaction. In the general case, when the number of label classes is unlimited, the problem is NP-hard. However, optimal partitioning can be obtained in linear time for the two class problem and effective heuristics exist for the general case.

Clearly, the key determinant of the effectiveness of the above procedure is the number of users using the search engine. As the number of users posing queries and clicking on search results increases, the coverage and the quality of the labels improve, leading to better ranker and better search results.

2.2 Diversity

A simple search box into which the user can type whatever string the user thinks best describes the information user is looking for has become a universal interface and the simplicity of this interface has been a key factor contributing to the immense popularity of web search. A consequence of this simplicity is that different users provide to the search engine the same query string even when they have very different information goals in mind. Joe might be looking for a hedge trimmer, whereas Dan might be looking for a beard trimmer, but both may ask the query – *trimmers*. While the users asking them have unique intents, the intent of many queries looks ambiguous to the search engine. How can then the search engine best answer such queries?

The approach taken in [5] is to diversify the search results so that the probability that an average user will find at least one relevant document in the retrieved result is maximized. Thus, a result page serves dual purpose - it offers relevant results for multiple intents of a given query and once the user has signaled a specific intent by a clicking on a document, further results take into account this disambiguation information.

The problem is formalized as follows. Given query *q*, a set of documents *D*, a probability distribution of categories for the query $P(c|q)$, the quality values of the documents $V(d|q,c)$, and an integer *k*, find a subset of documents *S* with $|S| = k$ that maximizes $P(S|q) = \sum_c P(c|q) (1 - \prod_{d \in S} (1 - V(d|q,c)))$. If $V(d|q,c)$ were interpreted as the probability that document *d* satisfies a user that issues query *q* with the intended category *c*, then the product term signifies the probability that the set of documents in category *c* will all fail to satisfy and one minus that product equals the probability that some document will satisfy category c. Finally, summing up over all categories, weighted by $P(c|q)$, gives the probability that the set of documents *S* satisfies the "average" user who issues query *q*.

The general problem is NP-hard, but the objective function admits a submodularity structure that can be exploited for the implementation of a good approximation algorithm. Without going into details, the overall idea is to probabilistically classify queries and documents into taxonomy of intents and then use the analogy of marginal utility to determine whether to include more results for an already covered intent. The classification is done by analyzing the log of documents clicked by different users asking queries on the search engine [6]. Again, the quality of results improve as more and more satisfied users ask even more queries and click on results.

3. BEYOND WEB SEARCH

Having described illustrative applications where data externality is already playing a role, we next give some speculative applications of data externality.

3.1 Health

An emergent social phenomenon in healthcare is that individuals are starting to take charge of their own health and trying to avoid needing care in the first place [7]. The tools they use include everything from pedometers/accelerometers that monitor footsteps and motion, to sleep monitors, pulse and heart monitors, and glucose monitors. New systems have started to emerge that will allow people to record everything pertaining to their life and to store all these data in a digital archive, possibly in the cloud [8].

This unprecedented creation of user data can lead to large positive externalities. A simple example is the ability to create demographic specific height and weight charts for infants.[1] Similarly, it has been now determined that the optimum

[1] When our daughter was growing up, her (height, weight) point always fell way below the chart. It was of very little comfort to me or my wife when we were told not to worry since the charts were developed for Caucasians.

cholesterol level for Asian Indians is 150 mg/dL (much lower than 200 mg/dL for Westerners) due to elevated levels of lipoprotein [9]. Such data-driven medical discoveries become feasible with the availability of archives of peoples' digital diaries. Of course, applications will have to be endowed with the right privacy and security capabilities for data sharing for such externalities to be realized.

3.2 Education

Quality of educational material plays a critical role in knowledge acquisition on part of students [10]. Consider an online learning setting in which the students are expected to take a multiple-choice quiz to self-test their understanding of the section. The scores for every question for every student are recorded. This data can now be potentially used to identify those sections of the textbook that might be confusing, and hence merit rewriting.

Postulate: Test score = f (student ability, clarity of material). We have record of test scores for a large number of students. Techniques such as those described in [11] can now be applied to estimate the clarity of the material. For another research effort directed at improving the quality of textbooks through data mining, see [12].

4. RESEARCH OPPORTUNITIES

We next sketch some research opportunities related to data externality.

Architecting for data externality. It is in the nature of externalities that they have tipping points where there is general acceptance and near-universal usage. Search engines are already seeing daily data rates in upwards of tens of terra bytes and the data rates are expected to continue to accelerate. Developing architectures for managing and handling such large data sets falls well within the purview of data researchers.

Privacy, security, confidentiality, trust. Data externality requires technology support for responsible data custodianship, without which the system will come unglued [13]. While the progress on this crucial topic will require further exploration of many streams of ongoing research, it may also provide opportunity for new approaches. For instance, an economic approach to privacy was hinted in [14, Section 7.2.4], which might be worth pursuing further.

Data mining and learning at scale. There is need for exploration along couple of complementary dimensions: How to scale the current solutions to work on qualitatively larger datasets? Do some techniques that were not competitive in past become competitive with the availability of much more data? Do we need to develop altogether new techniques?

Design principles. How does one identify applications that could be amenable to data externality? Are there general principles one must follow for taking advantage of data externality? How to ensure incentive compatibility between those providing data and those benefitting from data spillover?

Intrinsic value of data. The primary way the information services based on data externality become economically viable is from the value extracted by analyzing the spilled over data. It behooves then to think through how one can quantify the value of a collection of data or when can we say that one collection is more valuable than the other. This is terra incognita.

5. CONCLUSIONS

Information services developed to consciously take advantage of data externality is a relatively recent phenomenon. By definition, these services are data-intensive in nature. Most of them have been developed as one-ofs. Finding common abstractions, developing general algorithms, and architecting systems for supporting such services could constitute an exciting research agenda for data researchers.

6. ACKNOWLEDGMENTS

This paper is a synthesis of ideas and discussions in Search Labs.

7. REFERENCES

[1] Laffont, J. J. 2008. Externalities. *In The New Palgrave Dictionary of Economics*, S. N. Durlauf and L. E. Blume, Ed. Second Edition.

[2] Burges, C., Shaked, T., Renshaw, E., Deeds, M., Hamilton, N., and Hullender, G. 2005. Learning to Rank Using Gradient Descent. In *ICML*.

[3] Banko, M. and Brill, E. 2001. Scaling to Very Very Large Corpora for Natural Language Disambiguation. In *ACL*.

[4] Agrawal, R., Halverson, A., Kenthapadi, K., Mishra, N., and Tsaparas, P. 2009. Generating Labels from Clicks. In *WSDM*.

[5] Agrawal, R., Gollapudi, S., Halverson, A., and Ieong, S. 2009. Diversifying Search Results. In *WSDM*.

[6] Fuxman, A., Tsaparas, P., Achan, K., and Agrawal, R. 2008. Using the Wisdom of the Crowds for Keyword Generation. In *WWW*.

[7] Dyson, E. 2010. Heal Thyself. *Project Syndicate* (April 2010). DOI= http://www.project-syndicate.org/commentary/dyson19/English.

[8] Bell, G. and Gemmell, J. 2007. A Digital Life. *Scientific American* (March 2007).

[9] Enas et al. 2001. Coronary Artery Disease in Asian Indians. *Internet J. Cardiology.*

[10] Heyneman, S. P., Farrell, J. P., and Sepulveda-Stuardo, M. A. 1978. Textbooks and Achievement: What We Know. World Bank. PUB HG 3881.5 .W57 W67 No. 298.

[11] Baker, F. B., & Kim, S.-H. 2004. *Item Response Theory: Parameter Estimation Techniques* (2nd ed.). Marcel Dekker.

[12] Agrawal, R., Gollapudi, S., Kenthapadi, K., Srivastava, N., and Velu, R. 2010. Enriching Textbooks Through Data Mining. In *ACM DEV*.

[13] Federal Trade Commission 2010. Protecting Consumer Privacy in an Era of Rapid Change. December 2010.

[14] Agrawal, R., Freytag, J. C., and Ramakrishnan, R. (Eds.) 2004. Data Mining: The Next Generation. Dagstuhl Seminar 04292. July 2004.

No Bits Left Behind

Eugene Wu
MIT CSAIL
eugenewu@mit.edu

Carlo Curino
MIT CSAIL
curino@mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

One of the key tenets of database system design is making efficient use of storage and memory resources. However, existing database system implementations are actually extremely wasteful of such resources; for example, most systems leave a great deal of empty space in tuples, index pages, and data pages, and spend many CPU cycles reading cold records from disk that are never used. In this paper, we identify a number of such sources of waste, and present a series of techniques that limit this waste (e.g., forcing better memory locality for hot data and using empty space in index pages to cache popular tuples) without substantially complicating interfaces or system design. We show that these techniques effectively reduce memory requirements for real scenarios from the Wikipedia database (by up to $17.8\times$) while increasing query performance (by up to $8\times$).

1. INTRODUCTION

One of the core tenets of good database design is that resources, especially RAM and disk bandwidth, should be used efficiently. DBMS designers have used a number of techniques to optimize hardware utilization, such as keeping hot tuples in the buffer pool, and performing sequential access to the disk whenever possible. Every bit of memory or bandwidth used to fetch or store data that is not needed is a bit lost.

Despite this fact, many database implementations do a poor job of using these precious memory resources. For example, the typical index fill factors have been shown to be around 68% [10], and this figure can get much worse in the face of updates and deletes [6]. This 32% of space left unused is the product of a conscious design decision aimed at reducing expensive node split operations. Nonetheless, since index sizes often rival the size of the tables themselves, this is a very significant amount of memory dedicated to storing absolutely nothing.

Another example results from the use of fixed-size pages, which need to be read in their entirety just to fetch a single tuple. If the other tuples on a page are not needed, most of this I/O is completely wasted. Using a workload trace from Wikipedia, we found that 99% of accesses to Wikipedia's *revision* table, which stores metadata about article revisions, are focused on the 5% of tuples that represent the latest revision of the articles. The index clustering used in Wikipedia leads to heap pages that contain as little as 2% of frequently queried data.

A third example of waste has to do with inefficient encoding of data. This can arise due to poor choices in physical representation (e.g., a string representation for an integer, using bytes to store booleans or lack of compression) or semantic decisions (e.g., storing full timestamps when the application only requests years). We performed an analysis of tables on multiple databases and found between 16% to 83% of waste due to inefficient physical encoding.

In this paper, we describe these and several other examples of database system resource waste in more detail, and propose a number of solutions to these problems, including:

1. A technique to deal with the problem of wasted space in index pages, that "recycles" this space, using it as a cache for hot data. We show that the performance benefit we obtain on a substantial class of queries from the actual Wikipedia workload can be orders of magnitude higher, and that this technique can be implemented in a way that doesn't change index APIs or increase system complexity.
2. A technique to deal with the problem of cold tuples polluting pages containing hot data, that clusters commonly accessed tuples together, improving performance a factor of 6 while reducing total index sizes a factor of 19.
3. Analysis techniques that deal with the problem of encoding waste by identifying the most efficient type for a given column, and treating the programmer-supplied type merely as a declarative "hint" rather than an actual storage type. Additionally, we suggest ways to eliminate redundant data and to exploit ID fields by embedding semantic information in the values.

In summary, various artifacts of database system implementations, and poor user choices account for significant amounts of wasted storage in databases today. We believe there is a huge opportunity for research that i) optimizes DBMS structures and policies in a workload-specific manner, ii) empowers users with tools that automate waste detection and, iii) performs automatic space allocation and layout tuning. Work on self tuning databases [2] that reorganize column layouts [5], select optimal indexes [3], and calculate optimal database knobs [9] are high level optimizations in the right direction, however there is still ample opportunity at every level of the DBMS architecture, as we will illustrate in the rest of this paper.

This paper presents our *vision* to improve resource utilization in databases; as such, many of our experiments are simplified or incomplete—we plan to further extend this analysis in the future.

2. UNUSED SPACE

Unused space is defined as waste due to bytes that are allocated on disk or in RAM but *do not contain data*. Such waste is often allocated or reserved for expected data writes in the future, or as the result of poor data placement policies. We present one possible way to reuse this waste as it occurs in B+Tree pages.

2.1 Index Caches

The average fill factor of B+tree index pages is 68% [10]. In practice, real implementations are often substantially less – for example, in a frequently updated database for our CarTel (`http://cartel.csail.mit.edu`) research project, the fill factor is only 45%. The waste could be eliminated by compacting the pages with a fill factor of 100%, which would benefit read-only workloads, but lead to excessive node splits and result in low utilization of those pages in the presence of inserts. We want a way to use this space as a cache (that can be overwritten when the space is needed to store legitimate index contents); we describe

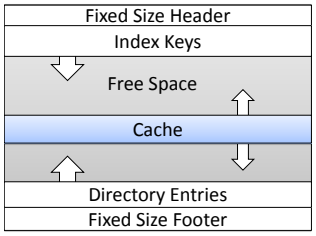


Figure 1: Anatomy of an index page

such a scheme in the rest of this section.

B+tree leaf nodes often store tuple pointers rather than physical tuple data, requiring an additional page access to retrieve the associated data page. In OLTP workloads, if the page access necessitates an additional disk I/O, the overhead can be substantial. By caching some tuples' data in their corresponding index leaf pages, the index can be used to directly answer queries, which makes use of unused space and avoids following pointers from leaf pages. We show experiments that suggest this can lead to substantial performance improvements on a trace of the Wikipedia workload without substantially complicating the index API or implementation.

As an alternative to a caching-based approach, one could imagine using covering indexes (i.e., adding all of the fields used in any query to the index key), which can also avoid accessing the heap to answer queries. However, covering indices still store cold data, waste space and bloat the index size, which wastes more total bytes, and increases pressure on RAM.

2.1.1 Cache Overview

The index cache stores field values of the most frequently accessed tuples in the index leaf pages, without introducing any disk I/O overhead. Most importantly, we preserve existing index maintenance algorithms and piggy-back off normal query processing to perform cache maintenance.

To simplify the discussion, we assume that the index keys and the tuples are fixed length. Metadata about the field lengths is stored on the index's metadata page in memory. The typical index page layout (Figure 1) consists of a fixed size page header and footer, a region of index key values, and a logically ordered directory of small fixed size pointers to the keys. The data and directory start from the high and low areas of the page and grow towards the center, and the page header stores pointers to the start and end of the free space, which is used as the cache store.

The cache space is split into slots where the beginning of each slot is aligned to the cache entry size (e.g., if the item size is 25 bytes, then the start of each slot is a multiple of 25). A slot either contains data or is zeroed out. Items begin with the tuple id to identify the data, followed by the field values.

When an index page is read during a lookup for a tuple with id t, we scan the cache slots for a cached version of t. Queries that project a subset of the index key and the cached fields can be answered without retrieving the data pages from the buffer pool, or worse, from disk (e.g., if the query projects A,B,C, the index key is A, B, and we cache C.) On a cache miss, we construct and insert the new cache item into the cache of the index leaf page that references the tuple, possibly evicting an existing item from that page's cache has not been accessed recently. In order to not introduce additional disk I/O, cache modifications do not dirty the page.

We avoid impacting existing index operations by allowing key inserts to freely overwrite the periphery of the cache space at any time. Thus it is important to keep hot items in the interior of the free space, where they will be overwritten last. It is possible to calculate the most stable location, S, as a function of (ignoring fixed size headers and footers) the index key size K, directory pointer size D, and page size P: $S = \frac{K}{K+D} \times P$

The cache is logically split into buckets of N slots each. When an item is first inserted, it is placed in a random free slot, or if no slots are free, evicts a random item in a peripheral bucket. On a lookup, we swap the item with a random entry in the adjacent bucket closer to S. This way, when the key and directory regions expand, the least accessed cache items are overwritten first.

2.1.2 Efficient Consistency Enforcement

So far, we have described how to read and populate the index cache. We now briefly outline an efficient index cache invalidation scheme to handle system crashes and modifications to a field whose cache page has been stolen. To support full index invalidation, we add a cache sequence number to each index page header (CSN_p), and maintain a global CSN for the index (CSN_{idx}). By preserving the invariants that 1) $CSN_p <= CSN_{idx}$ and 2) a page cache is only valid if $CSN_p = CSN_{idx}$, we can efficiently invalidate the entire cache by incrementing CSN_{idx}. Although this guarantees correctness, it is inefficient to invalidate the entire index on update queries. Instead, we create and store predicates that uniquely identify the updated tuples and append them to an in-memory log. When an index page is read during normal query execution, we zero the cache space if any predicates match keys in the page. If the list grows above a threshold, we can increment CSN_{idx} and clear the list. Finding a way to efficiently index and search these predicates is an area for future work.

2.1.3 Latching

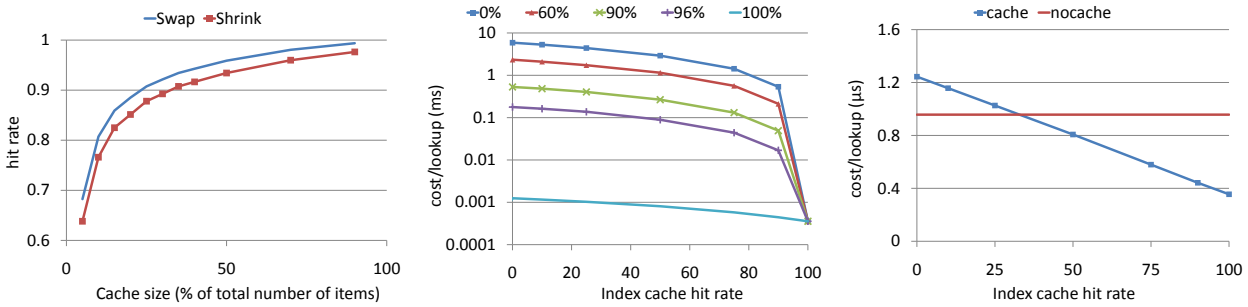
One point of concern is that we are turning every index leaf page read into a write, which may introduce higher lock/latch contention. Fortunately, normal index operations can freely overwrite the cache, so we only need to acquire short term latches for the duration of the cache writes. Additionally, we can give up a write operation if the latch is not immediately available.

2.1.4 Performance

In an analysis of Wikipedia, we found that the most popular class (40%) of queries accesses the *page* table using the *name_title* index, which uses a composite key of (namespace id, page title), and projects up to 4 additional fields. The index contains 360 MB of key data and, assuming that the index is 68% full and all 4 fields are cached (25 bytes/cache item), the index can store up to 7.9 million cache items – representing over 70% of the tuples in the *page* table and allowing us to answer nearly all of these queries through the index (due to skew in the page access).

We ran a simulation to study how the hit rate varies with the cache size using a zipfian distribution similar to Wikipedia ($\alpha = .5$) and found that the swapping based cache management algorithm exhibits high hit rates (Figure 2(a)). Each point is the average hit rate after 100k lookups and the x-axis is the percentage of the items that the cache can hold (which will vary depending on the tuple size and index fill factor). We compare $Swap$, which simulates a read-only workload that does not overwrite the index cache (constant cache size), and $Shrink$, which simulates a read/insert workload that overwrites half of the index cache at a constant rate over the duration of the experiment. $Swap$ exceeds 90% hit rate when the cache size is only 25% of the table. $Shrink$ only reduces the hit rate by 5%, showing that swapping effectively moves hot items towards the middle.

Figure 2(b) is a micro-benchmark that illustrates the performance benefits of index caching over a random lookup distribution. We assume that the index is fully in memory, and simulate the index and buffer pool using large in-memory arrays. An index cache miss must access a random page in the buffer pool, and a buffer pool miss must read a page from an on-disk file. We



(a) Hit rate as cache size varies, zipfian distribution ($\alpha = .5$)

(b) Query performance as index cache and buffer pool hit rates vary.

(c) Index cache performance with buffer pool hit rate = 100%

Figure 2: Index caching experiments

measure the performance improvement as the index cache (x-axis) and buffer pool (lines) hit rates vary from 0 to 100%. Different lines show what happens with increasing buffer pool hit rates. The hit rate depends on the available RAM for the buffer pool, as well as the sizes of the data pages and working set. Higher buffer pool hit rates, unsurprisingly, lead to better performance. The x-axis shows what happens as the hit rate of our index cache increases — higher cache hit rates substantially improve performance as we avoid both the memory access to the buffer pool as well as the additional disk I/O in the event that the data page is not in the buffer pool. Thus, even with when the dataset fully fits in memory (100% buffer pool hit rate), we still see a 2.7× improvement in performance by avoiding the additional memory accesses to pages in the buffer pool. We measured the index cache hit rate for our subset of the Wikipedia workload to be above 90%, suggesting that Wikipedia should see query performance gains of up to several orders of magnitude by employing this technique.

Figure 2(c) extends the previous experiment to illustrate the overhead of using index caching. We assume the buffer pool hit rate is 100% and see that index caching has 0.3us of overhead, which disappears when the cache hit rate exceeds 35%, and ultimately outperforms $nocache$ by 2.7×.

In our experiments, we hand picked the fields to cache in the index and discovered a number of useful heuristics for selecting these fields. First, the fields should be stable (i.e., rarely updated). Updates must access the updated field values in the heap tuple, so frequently updated fields do not benefit from index caching. Second, the cached fields should be chosen to fully answer a large class of queries. We found that over 40% of Wikipedia queries can be directly answered through an index cache on 4 attributes. These heuristics are at odds with each other, so the optimal choice of fields to cache is dependent on the workload, and is an interesting direction for future work.

2.2 Additional Directions

There are many other types of data that might be cached in index pages, for example: statistics, pre-computed query results, or other index pages are all options. In addition, these same concepts can be applied to data pages as well. For example, data pages can cache the results of foreign key joins, to avoid additional disk accesses for join queries. More ambitiously, if a data page is consistently read during the execution of a complex join query, caching the query results can offer substantial query performance improvements.

3. LOCALITY WASTE

We define locality waste as waste due to *ineffective placement* of data. A common example is when a full tuple must be read into memory even though the query only accesses a small subset of the fields. Equally wasteful is when cold tuples are read into memory only because they are co-located near a single hot tuple (e.g., on the same page). In this section we provide an example of access frequency based horizontal partitioning that improves query lookups in Wikipedia's *revision* table by over 8×, and sketch an improvement to index caching in update intensive scenarios using vertical partitioning.

3.1 Horizontal Partitioning

The first form of locality waste we discuss results from hot and cold tuples being co-located on the same data page. This occurs when the tuple placement strategy (e.g., append to table) does not match the access patterns. We argue for clustering and partitioning tables by access pattern rather than by range or hash partitioning. This is particularly beneficial for lookup based workloads that access a small set of hot tuples distributed throughout the table.

For example, Wikipedia's *revision* table stores a tuple for every new page revision; each tuple contains a unique revision ID, the page ID, a pointer to the text content, and several additional fields. 99.9% of page requests access the 5% of the tuples that represent the most recent revisions for each page; however, these hot tuples are scattered throughout the table, with as few as one hot tuple per data page (2% utilization). Hash and range partitioning are not possible because the access frequency is not related to any field values.

Figure 3 shows the possible performance benefits of access based clustering on Wikipedia's *revision* table. Our clustering algorithm relocates hot tuples by deleting then appending them to the end of the table. The 0%, 54%, and 100% curves vary the percentage of hot tuples that are clustered while the $Partition$ curve additionally creates a separate partition for the hot tuples. The workload is derived from 10% of 2 hours of Wikipedia's Apache logs. Lookup performance improved by 1.8× after clustering 54% of the hot tuples (2% of the table), and by 2.15× after clustering all hot tuples. Creating a partition for hot tuples reduces query costs by 8.4×. The reason partitioning has such a profound impact is that reducing the index size from 27.1 GB for the full table to 1.4 GB for the hot partition allows the entire index to fit in RAM.

The properties of the workload dictate how to identify hot tuples and move tuples between the hot and cold partitions. In Wikipedia, hot *revision* tuples are those that are pointed to from the *page* table, so newly inserted revision tuples can replace the previously hot tuple for the same page, which is then moved to the cold partition (note that this does require updating foreign key pointers and/or using forwarding tables to redirect queries using old ids to the new tuples). Other applications may have different policies, or require automated tools to keep track of access patterns.

3.2 Vertical Partitioning

Just as horizontal partitioning reduces locality waste, vertical partitioning can also be used to improve database performance [1, 7] and maximize memory utilization. For example, separating

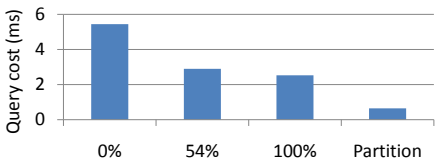


Figure 3: Cost per query

the cached fields from the uncached fields can complement index caching by minimizing the amount of redundant data read into memory when queries access fields not found in the index. Additionally, splitting the table based on the field update rate can increase the write density per page. Weighing the benefit of vertical partitioning against cost of merging the partitions together makes makes this problem non-trivial and interesting.

4. ENCODING WASTE

We define encoding waste as waste stemming from data that is *inefficiently represented*. This can be due to using inappropriate field types (strings for int values), poor compression, or more generally, storing data at a higher semantic granularity than what the application expects. For example, if the application stores timestamp values yet only expects years then the field contains encoding waste. In this section, we revisit the process of schema definitions and present techniques that exploit the semantic information in fields.

4.1 Automated Schema Optimization

Column values can be analyzed to understand the typical value range or the content properties (e.g., only numerical strings) and compare them against the declared types in the schema. Similarly, large fields that are either never accessed or only projected or accessed through equality predicates are good candidates for compression.

We analyzed several of the largest tables in the Cartel and Wikipedia databases and found that they can all reduce their physical encoding waste by 16% to 83% through simple techniques. For example, Wikipedia's *revision* table uses a 14 byte string to represent a timestamp that can easily be encoded into a 4 byte timestamp. Most commonly, we found a large number of int fields that store small value ranges which can easily be encoded in 8, or even 4 bits. Although individually small optimizations, the total amounted to over 23.5 GB (20%) of waste in the tables we inspected. Removing these unused bits increases the data density and consequently improves query execution performance.

We argue that schema type definitions should be treated as hints rather than hard constraints. Schemas are typically designed before the application is built when the workload is unclear. At this stage, it is safer to over-allocate space. However, as the database grows, the need for performance tuning, and the knowledge of field values and allocation requirements grows. At this point, automated tools can infer true field types and value distributions to modify internal field definitions and minimize encoding waste, or suggest these optimizations to the user.

4.2 Semantic IDs

The value of ID fields is often emantically meaningless to the application, other than as a unique identifier. In other words, the uniqueness, not the *value*, is important to the application. For example, Wikipedia and many applications that use object-relational-mapping (ORM) layers define an AUTO_INCREMENT primary key field for each table with such properties. Similarly, fields such as version number are used to induce order for a unique entity, whereas the actual value is inconsequential.

One response to this waste is reduction. Fields can be reduced if proxies exist whose values exhibit the same properties that the application expects. For example, ID fields representing uniqueness can be eliminated and the tuple's physical address can be used as a proxy. Column stores already infer the id using the tuple offset [8]. More generally, if there is a functional dependency $X \rightarrow Y$ and the semantic properties of Y can be directly inferred from X, then Y can be dropped.

The second response is to exploit the semantic opaqueness of primary ID fields and reassign the value to improve database performance. We propose embedding partition information directly in the ID field as a mechanism to implement the policy described in Section 3.1. If the data is clustered on the ID field, then simply updating the ID value is enough to physically move the tuple. Otherwise, the hot tuples can be shuffled to the end of the table by transactionally deleting and inserting the tuples.

This same idea can be applied to simplify query routing in distributed partitioned databases. Recent database partitioning work [4] attempts to find a partitioning that minimize the number of distributed transactions for a given workload. Unfortunately, this may require data placement at a per-tuple level, which necessitates a large routing table that maps tuple IDs to their physical location. Such tables can easily become a resource and performance bottleneck and limit the scalability of the routing infrastructure. Embedding a tuple's physical location in its ID alleviates this bottleneck and we believe it is an exciting area of future work.

5. CONCLUSIONS

In this paper, we highlighted three classes of waste in modern database systems—unused space, locality waste, and encoding waste. We suggested several techniques that have the potential to reduce or reuse this waste to dramatic effect. In particular, we proposed *access-based horizontal partitioning*, which groups hot tuples together, and *index caching*, which reuses unused space in B+Tree indexes, and showed that they can result in order-of-magnitude efficiency gains. In addition, we introduced the ideas of interpreting the schema as a layout hint, dropping implicit fields, and embedding information in fields based on application-level semantics. The encouraging results we obtained suggest that it may be time to revisit canonical designs (e.g., B+Trees with a 68% fill factor) in favor of more efficient ones, especially when (as we have shown) this efficiency can be obtained without impacting API or UI complexity.

6. REFERENCES

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In SIGMOD, 2004.
[2] S. Chaudhuri. Self-tuning database systems: A decade of progress. In in VLDB, 2007, pages 3–14.
[3] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In VLDB, 1997.
[4] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. PVLDB, 3(1), 2010.
[5] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In CIDR, 2007.
[6] T. Johnson and D. Shasha. Utilization of b-trees with inserts, deletes and modifies. In PODS. ACM, 1989.
[7] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In SSDBM, 2004.
[8] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In VLDB, 2005.
[9] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In VLDB, pages 20–31, 2002.
[10] A. C.-C. Yao. On random 2-3 trees. Acta Informatica, 9:159–170, 1978.

DBrev: Dreaming of a Database Revolution

Gjergji Kasneci
Microsoft Research
Cambridge, UK
gjergjik@microsoft.com

Jurgen Van Gael
Microsoft Research
Cambridge, UK
jvangael@microsoft.com

Thore Graepel
Microsoft Research
Cambridge, UK
thoreg@microsoft.com

ABSTRACT

The database community has provided excellent frameworks for efficient querying and online transaction or analytical processing. The main assumption underlying most of these frameworks is that there is no uncertainty regarding the stored data. However, in recent years, many important applications have emerged that need to manage noisy, corrupted, or incomplete data. This includes, e.g., anonymized data, data derived from sensor systems, or data from information extraction and integration systems. For such applications the assumption of logical consistency may not be valid and needs to be revised. In particular, techniques like probabilistic modelling and statistical inference may be necessary to be able to draw meaningful conclusions from the underlying data.

This paper presents DBrev, a hypothetical, intelligent database system for managing large quantities of data that involves uncertainty. We explain the main features of DBrev based on the scenario of information extraction and integration. We point out research challenges that need to be tackled and discuss a new set of assumptions that future database management frameworks need to build on.

1. INTRODUCTION

For many decades the Database (DB) community has focused on applications involving data that is not subject to uncertainty or where the uncertainty can be ignored or managed outside of the database. Such applications include accounting, payroll, inventory, etc. However, for a wide variety of recently emerged applications, uncertainty is abundant and unavoidable: in many applications, measurement reading from sensors can be corrupted, noisy or involve missing data; in applications dealing with anonymized data, uncertainty is part of the ambiguity arising from missing values in the data; and most prominently, in information extraction and integration, uncertainty comes from the imperfect automatic extraction and disambiguation techniques or from unreliable sources. It is widely recognized by the DB community that the capabilities and the relational-algebraic models offered by state-of-the-art DB management systems are not sufficient for applications such as the above [16, 17, 10, 15, 19]. Rather, for such applications, there is a need for DB systems that can automatically quantify uncertainty, resolve inconsistencies, and provide means for ranked retrieval and knowledge discovery for the stored data based on uncertainty. The main problems that such a system should be able to deal with are:

Provenance The system needs to be able to reason about the derivation process and the validity of the stored data, as well as about the reliability of the data sources.

Context Awareness The system needs to keep track of the context in which data is valid. This may involve inferring entities and categories from the data, as well as reasoning about temporal, spatial and other relevant context.

Ambiguity The system needs to maintain different context-dependent interpretations of data and support the disambiguation process at query time. This may involve inferring probabilities over interpretations, depending on context, and possibly notions of statistically inferred semantic similarity.

Consistency The system needs to maintain consistency beyond logical integrity constraints. This includes more complex (first-order logic) inference rules on the one hand and the handling of soft, probabilistic constraints on the other.

Searching and Ranking The system needs to provide ranked retrieval and knowledge discovery mechanisms that can quickly adapt to the search context, preferences, and needs of the user.

The above problems have been addressed in isolation by different communities, e.g, Databases, Machine Learning, Information Retrieval, etc., and can be approached by current techniques. However, addressing and solving them simultaneously in an integrated system is, from our point of view, an extremely challenging (and hence "outrageous") endeavor. The fundamental problem is to build the system on a framework for representing and updating beliefs under uncertainty. A promising candidate framework is probabilistic reasoning. Unfortunately, the scalable models used in state-of-the-art DB systems draw from first-order logic and are not designed to deal with probabilities. The Statistical Machine Learning (SML) community has given rise to comprehensive probabilistic reasoning models [20, 21, 22, 19], but these often still suffer from scalability issues. Jaynes' interpretation of probability theory as an extension of logic under uncertainty [21] points towards the commonalities of the DB and the SML communities. In this paper, we hypothetically join these two research avenues with the one of Information Retrieval, and present our dream system, DBrev, as their synergetic yield. As an example, we explain how DBrev helps constructing and maintaining large-scale knowledge bases containing billions of entity-relationship-entity triples (statements) extracted from the Web and other sources. DBrev enables probabilistic reasoning and provides ranked retrieval and knowledge discovery over the stored knowledge. In order to mitigate the uncertainty

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

inherent to information extraction and integration, DBrev aggregates statistics about different sources of evidence for the extracted triples, such as Web pages, extraction tools, Web 2.0 users, who may give feedback on the extracted triples, etc.

2. RELATED WORK

The main theoretical frameworks for combining the relational data representation with probabilistic reasoning are the *Probabilistic Database Model* and *Statistical Relational Learning*

Probabilistic Database Model (*PDM*) The PDM [16, 17, 10, 15] can be viewed as a generalization of the relational model which captures uncertainty with respect to the existence of database tuples (also known as tuple semantics) or to the values of database attributes (also known as attribute semantics). In the tuple semantics, the main assumption is that the existence of a tuple is independent of the existence of other tuples. Given a database consisting of a single table, the number of possible worlds (i.e. possible database instances) is 2^n, where n is the maximum number of the tuples in the table. Each possible world is associated with a probability which can be derived from the existence probabilities of the single tuples and from the independence assumption. In the attribute semantics, the existence of tuples is certain, whereas the values of attributes are uncertain. Again, the main assumption in this semantics is that the values attributes take are independent of each other. Each attribute is associated with a discrete probability distribution over the possible values it can take. Consequently, the attribute semantics is more expressive than the tuple-level semantics, since in general tuple-level uncertainty can be converted into attribute-level uncertainty by adding one more (Boolean) attribute. Both semantics could also be used in combination, however, the number of possible worlds would be much larger, and deriving complete probabilistic representations would be very costly. So far, there exists no formal semantics for continuous attribute values [16]. Another major disadvantage of PDMs is that they build on rigid and restrictive independence assumptions which cannot easily model correlations among tuples or attributes [10, 12, 19]. Such correlations, however, may be dictated by the application or domain at hand, and the underlying system has to provide a flexible framework to define and represent them.

Statistical Relational Learning (*SRL*) SRL models [12] are concerned with domains that exhibit uncertainty and relational structure. They combine a subset of relational calculus (first-order logic) with probabilistic graphical models, such as Bayesian or Markov networks to model uncertainty. These models can capture both, the tuple and the attribute semantics from the PDM and can represent correlations between relational tuples or attributes in a natural way [10]. More ambitious models in this realm are Markov Logic Networks [8, 19], Multi-Entity Bayesian Networks [13] and Probabilistic Relational Models [11]. Some of these models (e.g., [8, 13]) aim at exploiting the whole expressive power of first-order logic. While [8] represent the formalism of first-order logic by factor graphs, [11] and [13] deal with Bayesian networks applied to first-order logic. Usually, (approximate) inference in such models is performed using standard techniques such as belief propagation or Gibbs sampling. In order to avoid complex computations, [6, 7] propose the technique of lifted inference, which avoids materializing all objects in the domain by creating all possible groundings of the logical clauses. Although lifted inference can be more efficient than standard inference on these kinds of models, it is not clear whether they can be trivially lifted (see [9]). Hence, very often these models fall prey to high complexity when applied to practical cases. However, despite the complexity of probabilistic frameworks that build on graphical models, we think that future database systems can considerably benefit from lightweight graphical models for probabilistic reasoning, such as the ones presented in [14, 23].

Ranked Retrieval and Knowledge Discovery Finally, [4] and the references therein present approaches for combining Information Retrieval and Knowledge Discovery with current DB technology. Although the approaches discussed go a long way, they are rather static in nature by disregarding online updates of the data, which are inherent to many modern knowledge-oriented frameworks and applications, such as life-long information extraction [5], sensor networks and signal processing, etc. Most importantly, their frameworks do not consider holistic reasoning models for handling uncertainty.

3. DBREV

We illustrate the functionality of DBrev in the context of the management of information extracted from the Web. The system is continuously supplied with triples of the form <*entity*, *relationship*, *entity*>, where each triple comes with other metadata such as the URLs of Web pages from which it was extracted as well as temporal and/or spatial information about its validity (when available). In addition, DBrev continuously integrates implicit user feedback about the triples it contains; the feedback may be collected from an online game about encyclopedic knowledge or from users of Amazon's Mechanical Turk. The main tasks DBrev has to deal with are described in the following.

3.1 Data Provenance

The problem of data provenance (also known as the *lineage* problem) is closely related to the problem of database curation, which is an open problem in the presence of multiple information sources [16]. The idea is to trace the data derivation process back to the sources in order to guarantee data quality or to detect reasons for possible data inconsistencies. In probabilistic databases the lineage is handled by means of Boolean constraints on the tuples (e.g., *c-tables* [25]), which represent the set of possible worlds in which the tuples are true. In contrast, DBrev can compute the joint probability distribution over all possible worlds. Consequently, for any subset of triples, DBrev can return the maximum a posteriori assignment that maximizes their joint probability. Note that the triples can be related to each other through the sources they come from. Hence, DBrev constructs factor graphs in which the truth value of the triple is constrained by factors that relate it to variables quantifying the reliability of sources. This way the information sources become first-class citizens in DBrev. Furthermore, there can be other logical dependencies between the triples, such as dependencies concerning temporal and/or spatial dependencies [26]. These dependencies are translated into factor graphs as well, which are then integrated into the above factor graph. Consequently, they are handled as (soft) probabilistic constraints within the same reasoning framework (see Subsection 3.3). Efficient message passing on the factor graph corroborates the evidence and quantifies the uncertainty.

For example, consider the triple <*MichaelJackson*, *diedOn*, *25-07-2009*>. This triple could have been extracted from many different news pages and also from encyclopedic pages, such as Wikipedia. From a few other pages (e.g. `www.michaeljacksonsightings.com`), an extraction system could have extracted the triple <*MichaelJackson*, *seenIn*, *Cambridgeshire(UK)*> together with the temporal information '2010-03-08'[1]. In this case the corroboration process exploits temporal reasoning to decrease the truth value of the latter triple and the trust in `www.michaeljacksonsightings.com`. Note that the probabilistic corroboration problem is very subtle, as the truth values of triples and the trustworthiness of information sources

[1]In DBrev, temporal and spatial information about triples are represented by means of triple reification (see RDF Semantics at `http://www.w3.org/TR/rdf-mt/`).

are not necessarily determined by "majority voting" (e.g. by the number of Web pages or people who claim something, see also [23]). For example, if we corroborate user feedback about the triple <*BarackObama*, *hasWon*, *GrammyAward*> then a "majority voting" paradigm might fail since the majority of users may not know that Obama did indeed win the Grammy Award.

3.2 Ambiguity and Context Awareness

The ambiguity problem has been addressed in many variations, in different settings. It is one of the most acute problems in the field of information extraction and integration, where it arises in the form of *entity disambiguation/resolution*. For example, the integration of the datasets from different Social Web platforms, such as Facebook, MySpace, Twitter, flickr, LinkedIn, etc., poses a very hard problem, since the entities mentioned there can have ambiguous names. In the database setting, the problem occurs as the *record linkage* problem, where the goal is to find records that refer to the same entity. From a semantic point of view, the ambiguity problem is very difficult, as it often requires that contextual and background information be interpreted in the correct way. Hence, the ambiguity problem lies at the heart of AI. Consider the famous example sentence "The fruit flies like a banana". Contextual and background information play a decisive role for its understanding. At the same time, a probabilistic reasoning framework seems to be predestined for capturing the uncertainty that is inherent to disambiguation tasks.

For each entity, DBrev maintains two types of features: (1) ontological and (2) contextual features. While the contextual features are mainly provided by users and our extraction tools, the ontological features are automatically derived from general-purpose ontologies. For a given entity, the ontological features describe its taxonomic relations to other classes of entities (e.g. the entity *AlbertEinstein* belongs to the class *physicist*, *philosopher*, *person*, etc.). The contextual features consist of relevant terms (e.g. derived by frequency-based measures such as *tf-idf*) which occur in articles or user queries related to the given entity. While the ontological features represent some kind of commonsense background knowledge, the contextual features represent the different contexts that might be related to the given entity. The two types of features are combined into a unified representation and are used to map all the entities into a common latent space, in which the affinities or similarities between entities are measured. Similar ideas have been proposed in [24], where the authors describe a Bayesian model for the task of deriving feature-based similarities. On demand, the derived similarities allow DBrev to introduce for every pair of candidate entities e_1 and e_2 a new triple <e_1, *sameAs*, e_2>, which is assigned a corresponding probability (representing the belief that e_1 and e_2 are same) by the reasoning framework. This way, DBrev retains the flexibility to reassess its conclusions as new data comes in.

3.3 Consistency

In general, consistency can be viewed as a state (or possible world) in which a set of logical formulas are jointly satisfied. In databases, consistency is checked with respect to universal logical constraints (integrity constraints). A consistent transaction on a DB is one that does not violate those constraints. For example, the referential integrity constraints disallow dangling references, i.e., references to keys that do not exists in the DB.

DBrev exploits ontological knowledge, e.g. relationship properties, such as symmetry, transitivity, functionality[2], etc., to check whether the deductions between triples are consistent. Furthermore, as described in the previous subsection, DBrev combines the ontological knowledge with contextual knowledge to deal with ambiguity. The disambiguation component plays a critical role; without it the same entity might occur in the database in various dangling definitions, which would make logical deductions or transactions of any kind impossible. Consider the following rule, which describes the deduction of triples by exploiting the transitivity property of a relationship:

$$<X, R, Y> \wedge <Y, R, Z> \wedge$$
$$<R, type, TransitiveRelation> \rightarrow <X, R, Z>$$

where X, Y, and Z are entity variables, and R stands for a relationship variable. For example, from the triples <*MuséeDuLouvre*, *locatedIn*, *Paris*> and <*Paris*, *locatedIn*, *France*> DBrev can derive the triple <*MuséeDuLouvre*, *locatedIn*, *France*>. Although the latter triple may not be explicitly stored in the database, its derivation is very useful for the reasoning process, since it represents a logical constraint between triples. This is exploited to support the lineage (see Subsection 3.1) and the disambiguation (see Subsection 3.2). Consider a newly extracted triple <*"Louvre"*, *"is located in"*, *"France"*>. DBrev supports its disambiguation component by reasoning probabilistically about logical rules of the following kind:

$$refersTo(\text{“r”}, R) \wedge$$
$$refersTo(\text{“y”}, Y) \wedge$$
$$canBeDeduced(X, R, Y) \wedge D$$
$$\rightarrow refersTo(\text{“x”}, X)$$

where D represents a conjunction of contextual constraints (e.g., temporal, spatial, or domain-based constraints), R represents a relationship variable, and X and Y represent entity variables. This way DBrev can become more confident in the hypothesis that "Louvre" is a useful description for the entity *MuséeDuLouvre*. Similar rules were introduced in [1] to support the disambiguation process. However, DBrev allows users to define a wide range of logical constraints, which are interpreted as probabilistic rules (i.e., soft constraints) on the stored data; [23] shows how similar deduction rules can be translated into factor graphs.

3.4 Searching and Ranking

For large-scale information retrieval tasks such as web search, the ranking-oblivious conditions of Boolean search, which were mainly used for querying library or product catalogs, have been replaced by similarity and preference based ranking techniques involving vectorial or bag-of-words representations of documents and queries. Following the same trend, DBrev combines the unstructured conditions of keyword retrieval and the structured query paradigm of databases with question answering techniques, while making ranking a first-class citizen. This allows casual as well as expert users to query the system. The search and ranking model of DBrev is based on the following desiderata:

Pattern-Based Approximate Matching DBrev is geared to answer knowledge queries (i.e., queries that ask about entities and relationships between them) or questions. Knowledge queries can be expressed through a graph-based query language similar to the one proposed in [3]. An example search task could be: *"Find all US companies that are certified partners of Microsoft"*. Figure 1 depicts a graph-based representation of this query. The node labeled with $x represents a variable, which in the answering phase is replaced by entities that satisfy the relationship constraints given by the query graph. The expression *locatedIn** aims at capturing geographical hierarchies, e.g. cities, counties, states, countries, etc. Furthermore, node and edge labels are relaxed through labels that refer to the same entities and relations, respectively. For example, the node labeled "Microsoft" is relaxed through labels that might refer to the same real-world entity, e.g., "MS", "MS Corporation", "MSFT", etc. The entity disambiguation component of

[2]E.g. the relationship *X born on date Y* is functional, since every person can only have one date of birth.

DBrev takes care of retrieving similar labels for relaxation. Natural language questions are first translated to graph-based queries, which are then answered by means of the same relaxation technique.

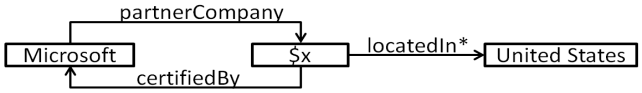


Figure 1: Graph-based representation of the search task "Find all US companies that are certified partners of Microsoft"

Top-k Ranking for Multiple Criteria Queries such as the above have a knowledge discovery character and may return a result set that is too large for a human to handle. This means that the results need to be ranked with respect to various criteria. The main criteria in DBrev are (1) similarity and (2) user preference. In an approximate matching paradigm, ranking by similarity (e.g. entity- or relationship-based similarity) is crucial. This allows DBrev to rank salient results higher than results that may be only vaguely related to the query. However, from a user perspective, the ranking becomes really meaningful if the system takes the user preferences into account. This is why DBrev makes use of the user context (e.g., location, background, general and current interests, etc.) and takes into account his information needs (e.g. information freshness, accuracy, popularity, etc.). Since the above criteria involve probabilities, which need to be aggregated in an efficient way, DBrev computes the results in a top-k fashion. This in lines with [18], where Ré et al. argue that in a probabilistic setting, the only meaningful semantics for returning results to a user is by ranking them. Finally, DBrev allows users to specify their own ranking criteria and provides hyperbolic visualization tools for data exploration.

4. CONCLUSION

In this "outrageous" paper we have speculated about a direction towards which database research may evolve. Our dream database system DBrev combines ideas from database research, machine learning and information retrieval to be able to manage the huge amounts of unreliable information extracted from the web. The challenge of large-scale information extraction illustrates how we need to employ and extend the notions of provenance, context, ambiguity, consistency, and ranking as key concepts for future database research. Although we have circumvented many other important questions (e.g. dynamic index updates, multidimensional indexing, etc.), we hope that the above mentioned research communities may take some inspiration from our dream and may seize the opportunity to collaborate on the challenges ahead.

5. REFERENCES

[1] Suchanek, F. M., Sozio, M., Weikum, G.: SOFIE: Self-Organizing Flexible Information Extraction. In: 18th International World Wide Web conference (WWW 2009), pp. 631–640. ACM Press (2009)

[2] Suchanek, F. M., Kasneci, G., Weikum, G.: Yago: A Core of Semantic Knowledge. In: 16th International World Wide Web Conference (WWW 2007), pp. 697–706. ACM Press (2007)

[3] Kasneci, G., Suchanek, F. M., Ifrim, G., Ramanath, M., Weikum, G.: NAGA: Searching and Ranking Knowledge. In: 24th International Conference on Data Engineering (ICDE 2008), pp. 953–962. IEEE (2008)

[4] Weikum, G., Kasneci, G., Ramanath, M., Suchanek, F.: Database and Information-Retrieval Methods for Knowledge Discovery. In.: Communications of the ACM (CACM 2009), pp. 56–64. ACM Press (2009)

[5] Banko, M., Etzioni, O.: Strategies for Lifelong Knowledge Extraction from the Web. In: 4th International Conference on Knowledge Capture (K-CAP 2007), pp. 95–102. ACM Press (2007)

[6] Poole, D.: First-Order Probabilistic Inference. In: 8th International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 985–991, Morgan Kaufmann (2003)

[7] Domingos, P., Singla, P.: Lifted First-Order Belief Propagation. In: 23rd AAAI Conference on Artificial Intelligence (AAAI 2008), pp. 1094–1099. AAAI Press (2008)

[8] Domingos, P., Richardson, M.: Markov Logic Networks. In: Machine Learning, 62(1–2), pp. 107–136. Springer (2006)

[9] Jaimovich, A., Meshi, O., Friedman, N.: Template Based Inference in Symmetric Relational Markov Random Fields. In: 23rd Conference on Uncertainty in Artificial Intelligence (UAI 2007), pp. 191-199. AUAI Press (2007)

[10] Sen, P., Deshpande, A., Getoor, L.: PrDB: Managing and Exploiting Rich Correlations in Probabilistic Databases. In: Journal of Very Large Databases, 18(5), pp. 1065–1090. Springer (2009)

[11] Friedman, N., Getoor, L., Koller, D., Pfeffer, A. Learning Probabilistic Relational Models. In: 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), pp. 1300–1309. Morgan Kaufman (1999)

[12] Getoor, L.: Tutorial on Statistical Relational Learning. In: 15th International Inductive Logic Programming Conference (ILP 2005), Springer (2005)

[13] Da Costa, P. C. G., Ladeira, M., Carvalho, R. N., Laskey, K. B., Santos, L. L., Matsumoto, S.: A First-Order Bayesian Tool for Probabilistic Ontologies. In: 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS 2008), pp. 631–636. AAAI Press (2008)

[14] Wang, D. Z., Michelakis, E., Garofalakis, M., Hellerstein, J. M.: BayesStore: managing large, uncertain data repositories with probabilistic graphical models. In: 34th International Conference on Very Large Data Bases (VLDB 2008), 1(1), pp. 340–351, ACM Press (2008)

[15] Antova, L., Koch, C., Olteanu, D.: 10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information. In: 23rd International Conference on Data Engineering (ICDE 2007), pp. 606–615. IEEE (2007)

[16] Dalvi, N. N., Ré, C., Suciu, D.: Probabilistic Databases: Diamonds in the Dirt. In: Communications of ACM, 52(7), (CACM 2009), pp. 86–94. ACM Press (2009)

[17] Agrawal, P., Benjelloun, O., Sarma, A. D., Hayworth, C., Nabar, S. U., Sugihara, T., Widom, J.: Trio: A System for Data, Uncertainty, and Lineage. In: 32nd International Conference on Very Large Data Bases (VLDB 2006), pp. 1151–1154. ACM Press (2006)

[18] Ré, C., Dalvi, N., Suciu, D.: Efficient Top-k Query Evaluation on a Probabilistic Database. In: 23rd Very Large Databases Conference (VLDB 2007), pp. 51–62. ACM Press (2007)

[19] Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning. MIT press (2007)

[20] Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1997)

[21] Jaynes, E. T.: Probability Theory – The Logic of Science. Cambridge University Press (2003)

[22] Bishop, C. M.: Pattern Recognition and Machine Learning. Springer (2007)

[23] Kasneci, G., Gael, J. V., Herbrich, R., Graepel, T.: Bayesian Knowledge Corroboration with Logical Rules and User Feedback. In: ECML PKDD 2010, Springer (2010)

[24] Stern, D., Herbrich, R., Graepel, T.: Matchbox: Large Scale Bayesian Recommendations. In: International World Wide Web Conference WWW 2009 (2009), pp. 111–120, ACM Press (2009)

[25] Imielinski, T., Lipski, W.: Incomplete Information in Relational Databases. In: Journal of the ACM, 31, pp. 761–791, ACM Press (1984)

[26] Gérard Ligozat, G., Mitra, D., Condotta, J.: Spatial and temporal reasoning: beyond Allen's calculus. In: AI Communications, 17(4), pp. 223-233, IOS Press (2004)

Towards a One Size Fits All Database Architecture*

[Outrageous Ideas and Vision track]

Jens Dittrich Alekh Jindal

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

We propose a new type of database system coined OctopusDB. Our approach suggests a unified, *one size fits all* data processing architecture for OLTP, OLAP, streaming systems, and scan-oriented database systems. OctopusDB radically departs from existing architectures in the following way: it uses a logical event log as its primary storage structure. To make this approach efficient we introduce the concept of *Storage Views (SV)*, i.e. secondary, alternative physical data representations covering all or subsets of the primary log. OctopusDB (1) allows us to use different types of SVs for different subsets of the data; and (2) eliminates the need to use different types of database systems for different applications. Thus, based on the workload, OctopusDB emulates different types of systems (row stores, column stores, streaming systems, and more importantly, any hybrid combination of these). This is a feature impossible to achieve with traditional DBMSs.

1. INTRODUCTION

1.1 Background and Motivation

In the past ten years we have seen considerable evidence that there is no *one size fits all* database architecture. We are currently witnessing a split of data management systems into several specialized solutions [20, 21]. For instance, for data warehousing database engineers already understood in the mid-nineties [9] that the DBMSs of that time were ill-equipped to cope with the size of the datasets and complexity of OLAP-queries. Therefore a separate type of system was forked from the one size fits all DBMS code line [10]. That system is based on a column store and became one of the most popular and successful approaches for OLAP; products include SAP BI Accelerator, InfiniDB, Paraccel. At the same time other types of systems were forked including DSMS (data stream management systems) [23]; products include StreamBase.

1.2 Research Problem

As a consequence, today's companies have to manage and integrate several types of data management systems. Data has to be copied from one database system to another. To achieve this, complex, ETL-style data pipelines have to be glued together. The different database systems may also use different query languages or dialects. Obviously, all of this leads to extra costs in terms of development costs, maintenance costs, and DBA costs.

So rather than making the world of data management easier, we have created a *zoo* of systems that sometimes has the opposite effect: it makes life of a company harder and more costly. We agree that for companies who invest a lot into connecting the different species in their zoo, it will eventually lead to a well-integrated and efficient overall system. Still, we believe that the zoo-keeping costs are non-trivial, especially for small to medium-sized business. We also believe that adapting such a zoo to new requirements, changing workload, or new types of applications may be prohibitive. Thus, the research challenge is to build a single efficient system covering the different database use-cases.

2. OCTOPUSDB

In this paper we take a radically new approach: we propose a single type of database system coined *OctopusDB*[1] that is able to mimic the behavior of the different species in the zoo.

Core Idea. The main idea of our system is to drop the assumption that a database system is developed around a central store (be it a row, column, or any hybrid store such as PAX [1] or fractured mirrors [17]) along with an ARIES-style [15] recovery log. OctopusDB does not have a fixed store. In OctopusDB all data is collected in a central log, i.e., all insert and update-operations create logical log-entries in that log. Based on that log we may then define several types of optional *Storage Views*. A *Storage View (SV)* represents all or part of the log in a different (or the same) physical layout. OctopusDB creates SVs transparently and solely based on the workload — and not based on some static decision for a concrete database product and hence a concrete storage layout. This single abstraction has another interesting consequence: the query optimization, view maintenance, index selection, as well as the store selection problems suddenly become a single problem: *storage view selection*, which OctopusDB treats inside a single *holistic storage view optimizer*.

The remainder of this section introduces OctopusDB's data model, primary log, and system interface. Section 3 introduces the concept of Storage View (SV). Section 4 introduces OctopusDB's holistic SV optimizer, Section 5 describes log purging and checkpointing, Section 6 discusses recovery and Section 7 explains con-

*Patent Pending - applied for by Saarland University [7].

[1] An octopus may adapt to its surroundings using a camouflage unmatched by any other species on earth: it may change both the color *and* the texture of its skin. Additionally, some octopus species may even mimic movements and shape thus *impersonating* other species, e.g. http://marinebio.org/species.asp?id=260

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

current transaction isolation in OctopusDB.

2.1 Data Model

The data items managed by OctopusDB are tuples $t_i = (a_1, .., a_{n(i)}),\ 0 \leq i \leq N$ where attributes $a_1, \ldots, a_{n(i)}$ may be of any type. The number of attributes for tuple i is given by $n(i)$. Each tuple is associated to a bag and a key. The bag is used to define subsets of the tuples, e.g. tables, partitions, collections. key identifies a tuple inside a bag. For simplicity, we assume a relational model throughout this paper. Therefore all tuples having the same *bag* share the same set of attributes (=schema).

2.2 The Primary Log Store

OctopusDB does *not* keep a row-, column- or any other store by default. All calls to the system interface are simply recorded in a sequential log, called *primary log*, creating appropriate logical log records. The primary log is itself an SV. OctopusDB stores its log persistently on durable storage (hard disk or SSD) following the write-ahead logging-protocol (WAL). For efficiency reasons we may keep a copy of the log in main memory, however this is no requirement. Each call to the system interface of OctopusDB internally creates a log record with an associated log sequence number *lsn*. As in traditional DBMSs no two log records may have the same lsn, therefore entries to the log are serialized. For the moment, all log records are *logical* and represent a *new state* defined by an operation[2]. Therefore, in contrast to ARIES, our log records do *not* represent changes that have been or should be applied to the database store. Our log simply contains the event history of operations without specifying how these events map to a particular store[3]. Thus, the format of our log record is (lsn, <method>, <parameters>) where <method> denotes the method of the system interface called and <parameters> denotes the parameters passed.

2.3 System Interface

Apart from traditional DBMS components (transaction manager, query optimizer, etc.), OctopusDB has a *primary log store* (to store the primary log) and a *storage view store* (to store additional storage views). To operate these, OctopusDB has a simple yet powerful interface containing the following methods:

registerSV(String svID, Type svType<, additionalPar>): creates and registers an SV of type svType having a unique identifier svID. Additional parameters may be passed to the SV.

registerQuery(String queryID, Query Q<, callback>): registers a query having a unique identifier queryID. An additional callback function may be passed.

snapshot(String outputSVID, String queryID): computes the result of the query and materializes it into the output SV.

maintain(String ouputSVID, String queryID): Same as snapshot, however, future updates will be reflected in outputSVID.

drop(String ID): Drops a query or SV from the system.

query(Query Q)→ Iterator it: Queries and/or modifies data in OctopusDB as specified in Q.

iterate(String ID) → Iterator it: Returns an iterator over the contents of the given query or SV.

Query definitions may be either a relational algebra expression as suggested in [4], SQL, or Pig Latin [16]. We will assume a relational algebra expression throughout this paper.

[2]In addition, transitional log records may be used, e.g. $a = a + 42$.

[3]The major performance advantage of ARIES is that it is using *physical* logging for REDO, i.e. intertwining a particular store with the log is a *feature* of ARIES. However, this feature may also be implemented in OctopusDB without giving up the logical primary log. See Section 6 for details.

3. STORAGE VIEW EXAMPLES

Storage Views (SVs) allow us to define arbitrary physical representations on the log. The main idea is to store the entire or a subset of the log or any other SV using a different physical layout. SVs *always materialize* their data. In general we create a network of SV dependencies with the goal to balance update and query processing costs. The dependency graph between different SVs is called the *SV lattice*. It is similar to the one used in materialized view maintenance (e.g. in data warehouses). However, the SV lattice is more general as it is not restricted to queries only but also has to consider the underlying storage layout. The interface to a SV contains the following private methods:

iterate(String queryID)→ Iterator it: Returns the result of the given query as an unordered iterator it. The query must be restricted to data covered by this SV.

iterationCost(String queryID)→ Cost c: returns the estimated cost c of the given query. In other words, it estimates the cost of **iterate(String queryID)**.

transformationCost(Type svType)→ Cost c: returns the estimated cost c for transforming this SV into svType.

For the flight booking use-case having tickets and customers data, presented in [25] and evaluated in the context of data layouts in [13], we shall now incrementally show the self-adapting API calls made by the system.

Log SV. Initially, the SV store does not contain any SVs; it only contains a single registered join query over tickets and customer data. It would be evaluated by scanning the primary log. However, as the log becomes too big, the system splits it into two logs:

```
registerSV("ticketsLog",LogSV);
registerSV("customersLog",LogSV);
registerQuery("customersOnly",σ_{bag=customers});
registerQuery("ticketsOnly",σ_{bag=tickets});
maintain("ticketsLog","ticketsOnly");
maintain("customersLog","customersOnly");
```

Further, OctopusDB consolidates the different versions for the same `(bag,key)`-pair to only keep the most recent one:

```
registerQuery("custRecent",γ_{recent}(Γ_{bag,key}(customersOnly)));
registerQuery("tickRecent",γ_{recent}(Γ_{bag,key}(ticketsOnly)));
maintain("ticketsLog","tickRecent");
maintain("customersLog","custRecent");
```

Row, Col SVs. The main idea of a Row SV, resp. Col SV, is to create a row store, resp. column store (or both), for any given table. Further, our system can create Row, Col SVs for any static or dynamic partition of the tables. For instance, OctopusDB creates *hot* and *cold* SVs for 7 day query window as follows:

```
registerSV("ticketsCold", ColSV);
registerSV("ticketsHot", ColSV);
registerQuery("tickRecentHot",σ_{time≥now−7days}(tickRecent));
registerQuery("tickRecentCold",σ_{time<now−7days}(tickRecent));
maintain("ticketsCold", "tickRecentCold");
maintain("ticketsHot", "tickRecentHot");
drop("ticketsLog");
```

Index SV. Indexes like B^+-trees, hash indexes, bitmaps, cache-optimized-trees, R-trees, inverted indexes, and so forth are just another type of SVs. The index may also be build on only parts of a table thus mimicking partial indexing [19]. In our use-case, OctopusDB build Index SVs over customers and hot tickets as follows:

```
registerSV("ticketsHotIndex",IndexSV,uncl,key=price);
registerSV("customersIndex",IndexSV,cl,key=id);
registerQuery("tickI1",π_{price,rid}(ticketRecentHot));
registerQuery("custI2",π_{ID,rid}(custRecent));
maintain("ticketsHotIndex","tickI1");
maintain("customersIndex","custI2");
```

In summary, the storage view concept allows us to model several important data managing concepts using a single abstraction only: both types of queries (point-in-time and continuous queries); different database stores (row-, col-, hybrid, etc.); and also the traditional query views (dynamic or materialized).

4. HOLISTIC SV OPTIMIZER

In general, each class of SV may implement its own access algorithms optimized for the particular storage structure. For instance, a Row SV may use row-wise compression and row-oriented iteration, e.g. [11]. In contrast, a Col SV may implement column-oriented compression and vectorized iteration [2]. Outside those SVs OctopusDB's *Holistic Storage View Optimizer* then implements any appropriate techniques for storage view selection, update propagation and query processing. To perform these, OctopusDB has three *cost models* for Log, Row, Col and Index SVs: (1) A *query cost model*, which models the random and sequential I/O costs, (2) an *update cost model*, which models the minimum of chunk/random update costs, also considering the leaf/node split costs in Index SVs, and (3) a *transformation cost model*, which models the costs to transform one type of SV to another. Below, we briefly describe some of the optimization features in Holistic SV Optimizer.

SV Rearrangement. The holistic SV optimizer can rearrange the SV lattice in order to balance query and update costs. This implies that the SV optimizer decides how to connect the *tail* of an arrow to the existing SV lattice. One particular advantage of using a holistic optimizer is that the query operators can be pushed down through the entire SV store. — even beyond the primary log.

Operator Log-Pushdown. In certain situations, the optimizer may decide to push down some of the selections and projections as follows: (1) we examine the projections of all registered queries and compute the union set of attributes, (2) we push these projections down the lattice until the primary log. Similarly, for selections we (1) compute a conjunctive selection, and (2) push it even beyond the primary log. This means that any incoming log record will be checked even *before* putting it into the Log SV in the primary log.

Adaptive Partial SVs. The holistic SV optimizer can inject additional SVs to speed-up query processing. For instance, it does not make sense to build an index for an entire relation if only parts of that relation are queries. Techniques such as partial indexing [19] can be extended to create *dynamic* partial SVs.

Stream Transformation. For applications having continuous queries, we may only select a *window* of interest over the *unbounded* stream of log records i.e. the primary logical log in OctopusDB. This means the "database store" simply consists of several windows of interest. No other (older) data needs to be kept. OctopusDB can mimic this as follows: (1) do not use a Log SV for the Primary Log Store. (2) route all incoming log records to all relevant queries, (3) push possible updates up the SV lattice. In other words, we are reducing the stream processing problem to a *SV maintenance problem*.

Other Use-Cases. By creating the right SVs OctopusDB can mimic a variety of system e.g. OLTP, OLAP, Streaming Systems. Furthermore, by combining different SVs OctopusDB can emulate newer hybrid systems, for instance combinations of continuous and store (=archival) queries which has been researched heavily in the past years [6]. Table 1 lists several use-cases for OctopusDB.

5. PURGING AND CHECKPOINTING

The primary log may eventually grow too large, especially if the update rate is too high or the database has been up for a while and collected a long log of change operations. In this situation we need to shrink the size of the log. There are several options:

Purge log records for data that is not of interest anymore, e.g. changes older than two years are not needed for OLTP apps.

Compress the log, thus saving the storage space.

Checkpoint i.e. write a *begin checkpoint log record* to the log, create a storage view for all log records older than the begin checkpoint log record, and finally write an *end checkpoint log record*.

Use-Case (traditional systems)	Storage view definition type	Storage view definition example query
row store	Row SV	any
column store	Col SV	any
PAX	PAX SV	any
fractured mirrors	Row SV and Col SV	same query for both
column groups	Row SV and Col SV	$\pi_{a_1,\ldots,a_k}$ $\pi_{a_{k+1},\ldots,a_m}$
index	Index SV	any
indexed row store	Index SV(Row SV)	any
indexed column store	Index SV(Col SV)	any
read-optimized indexed column store **+ differential write-optimized row store**	Index SV(Col SV) Row SV	$\sigma_{t<\text{now}()-1\text{day}}$ $\sigma_{t\geq\text{now}()-1\text{day}}$
partial index	Index SV	$\sigma_{420\leq a_k\leq 42000}$
projection index	Col SV	π_{a_k}
partial projection index	Index SV(Col SV)	$\pi_{a_k}\left(\sigma_{420\leq a_k\leq 42000}\right)$
DSMS	Index SV	$\sigma_{t\geq\text{now}()-5\text{min}}$
DSMS **+ archive**	Index SV and Col SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-5\text{min}}$
snapshot	any	any
replicated row store	Row SV Row SV	same query for both
query	any	any
dynamic view	any	any
materialized view	any	any

Use-Case (new system)	Storage view definition type	Storage view definition example query
OLTP **+ OLAP**	Row SV Col SV	$\sigma_{t\geq\text{now}()-1\text{day}}$ $\sigma_{t<\text{now}()-1\text{day}}$
DSMS **+ OLTP**	Index SV Row SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-5\text{min}}$
DSMS **+ archive OLTP** **+ archive OLAP**	Index SV Row SV Col SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{\text{now}()-1\text{day}\leq t<\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-1\text{day}}$
other hybrid	any combination of the above	any

Table 1: Use-Cases of OctopusDB

Then we purge all log records older than the begin checkpoint log record. Depending on the storage view we use for a checkpoint, we can *(a) archive:* Use a RowSV or ColSV, *(b) aggregate:* aggregate part of the log, or *(c) re-checkpoint:* Replace an existing checkpoint in the log with a derived checkpoint.

6. RECOVERY

Logical Recovery. Recovery depends on the purging strategy used. For no purging or checkpointing, since OctopusDB keeps the primary log on durable storage, simply copy the log from durable storage to main memory and OctopusDB is fully recovered. In the background OctopusDB will then re-create all SVs that existed before the crash. Note that the recovery process does not have to put any information on the progress information into the log, e.g. like compensation log records in ARIES [15]. This substantially simplifies the code base of our system. In case the log is purged or checkpointed (i.e. incomplete), we read the log sequentially starting from the oldest entry and collect begin checkpoint log records into a *checkpoint* set. If we find an end checkpoint log record then we remove the corresponding begin checkpoint from the set. If after reading the log *checkpoint* is empty, we proceed as if no log purging or checkpointing ever happened. Otherwise we copy the log to main memory, however ignore all checkpoints missing an end checkpoint log record. After copying this partial log, OctopusDB is recovered. After that, in the background we re-create all checkpoints that did not have an end checkpoint log record.

ARIES-style Physiological Recovery. One might argue that OctopusDB's recovery algorithm gives away some performance by not using page-oriented (physiological) REDO as in ARIES[4]. However, OctopusDB could easily be extended to keep physiological redo information as well. The trick is to write physiological REDO information for each SV separately. UNDO may still be performed using the global logical log. Conceptually, this algorithm then does not differ from ARIES anymore.

7. TRANSACTIONS AND ISOLATION

To support concurrent execution of transactions we extend OctopusDB's system interface by three methods:

beginTA()→ taID: starts a transaction.
commitTA(taID)→ bool: commits transaction.
abortTA(taID)→ bool: aborts transaction.

Furthermore, we extend the methods of OctopusDB's system interface (Section 2.3) to receive an additional taID parameter. Thus, we may define arbitrary transaction sequences. Now let's discuss how to achieve ACID in OctopusDB. As in DBMSs, *Consistency* may be guaranteed by validating a set of integrity constraints at commit time.The *Isolation* algorithm of OctopusDB is a variant of optimistic concurrency control. Its core idea is to append all changes (uncommitted or committed) to the primary log but only to propagate committed data to any secondary SVs. Uncommitted transactions can *read* committed data either from the log or any secondary SV. They are allowed to *write* any data object they desire by adding log records to the log, but: the latter modifications are *not* yet propagated to the secondary SVs. Using this approach *Atomicity* is trivial as only transactions having a commit log record are reflected in a SV and need to be considered by other operations. The same holds for *Durability*: as mentioned before, OctopusDB follows WAL anyway. Since our log records are not condensed into a store in the first place (as in current DBMSs), we do not need undo, redo, before or after images of pages, nor compensation log records to achieve idempotency. Finally, since SV update propagation process is a possible synchronization bottleneck, it could be interesting to improve this to enable eventual or timeline consistency *among* storage views, i.e. trade consistency for performance.

8. RELATED WORK & CONCLUSION

Several authors have supported the idea of different types of database systems for different markets/use-cases [9, 20, 21], splitting the landscape into at least four different systems: SearchEngines (read-only inverted index), OLTP (transactional row store), OLAP (read-only column store) and DSMS (continuous window queries on unbounded streams). OctopusDB is not restricted to a particular store and workload and hence there is no OLTP/OLAP boundary. OctopusDB extends this seamlessness further to DSMS. Due to the dramatic changes in hardware, HStore [22, 14] as well as several scanning techniques [11, 18, 3] propose stripped down or simplistic versions of traditional DBMSs. The design of OctopusDB is simple by default and adds only as much complexity as really needed. Rodent store [5] allows DBAs to declare the database store using an algebra and GMAP [24] presents a DDL for defining physical structures. However, in contrast to OctopusDB, these either still assume a fixed store or do not handle unification with streaming systems, automatic store selection and workload adaption. Cracked databases, e.g. [12], similar to partial indexing [19] and adaptive indexing [8], break database tables into horizontal pieces by piggy-backing index-reorganization requests to individual queries. However, in contrast to OctopusDB, cracked databases assume a fixed column store. Finally, MySQL allows users to plugin application specific custom storage engines. However, they are statically configured and offline installed by an administrator. In contrast, OctopusDB not only creates storage views adaptively over the application life cycle, but can also store any subset of the data in any arbitrary physical representation.

[4]Note that UNDO is logical in ARIES anyway.

Conclusion. This paper opened the book for one-size-fits-all database architecture. We presented OctopusDB as a single system for OLTP, OLAP, streaming databases, as well as several other types of databases. With OctopusDB we are inverting the traditional DBMS development philosophy: a specific store, which is an irrevocable design-decision, built-in into the DBMS and an ARIES-style [15] log-based recovery implemented on top. Instead, in our approach we start with the log (which is totally disconnected from any store) in the first place and if necessary, we define optional SVs on that log suited for a particular workload.

Acknowledgment. Work partially supported by MMCI.

9. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.
[2] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.
[3] G. Candea et al. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *PVLDB*, 2009.
[4] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB*, 2000.
[5] P. Cudré-Mauroux et al. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.
[6] N. Dindar et al. DejaVu: declarative pattern matching over live and archived streams of events (Demo). In *SIGMOD*, 2009.
[7] J. Dittrich and A. Jindal. A method for storing and accessing data in a database system. Patent Application, 2010.
[8] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: adaptive indexing for context-aware information filters. In *SIGMOD*, 2005.
[9] C. D. French. "One size fits all" database architectures do not work for DSS. In *SIGMOD*, 1995.
[10] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *ICDE*, 1997.
[11] A. L. Holloway et al. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *SIGMOD*, 2007.
[12] S. Idreos and otherts. Database Cracking. In *CIDR*, 2007.
[13] A. Jindal. The Mimicking Octopus: Towards a one-size-fits-all Database Architecture. In *VLDB PhD Workshop*, 2010.
[14] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System (Demo). In *PVLDB*, 2008.
[15] C. Mohan et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, 1992.
[16] C. Olston et al. Pig Latin: a Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[17] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.
[18] V. Raman et al. Constant-Time Query Processing. In *ICDE*, 2008.
[19] M. Stonebraker. The Case For Partial Indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.
[20] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, 2005.
[21] M. Stonebraker et al. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR*, 2007.
[22] M. Stonebraker et al. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, 2007.
[23] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, 1992.
[24] O. G. Tsatalos et al. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB J.*, 5(2):101–118, 1996.
[25] P. Unterbrunner et al. Predictable Performance for Unpredictable Workloads. In *PVLDB*, 2009.

Longitudinal Analytics on Web Archive Data: It's About Time!*

Gerhard Weikum[1], Nikos Ntarmos[2], Marc Spaniol[1], Peter Triantafillou[2], András Benczúr[3], Scott Kirkpatrick[4], Philippe Rigaux[5], and Mark Williamson[6]
Max Planck Institute for Informatics, Germany[1]
University of Patras, Greece[2]
Hungarian Academy of Sciences, Hungary[3]
Hebrew University, Israel[4]
Internet Memory Foundation, France[5]
Hanzo Archives Ltd., UK[6]
weikum@mpi-inf.mpg.de ntarmos@ceid.upatras.gr mspaniol@mpi-inf.mpg.de peter@ceid.upatras.gr
benczur@sztaki.hu kirk@cs.huji.ac.il philippe.rigaux@internetmemory.org markw@hanzoarchives.com

ABSTRACT

Organizations like the Internet Archive have been capturing Web contents over decades, building up huge repositories of time-versioned pages. The timestamp annotations and the sheer volume of multi-modal content constitutes a gold mine for analysts of all sorts, across different application areas, from political analysts and marketing agencies to academic researchers and product developers. In contrast to traditional data analytics on click logs, the focus is on longitudinal studies over very long horizons. This longitudinal aspect affects and concerns all data and metadata, from the content itself, to the indices and the statistical metadata maintained for it. Moreover, advanced analysts prefer to deal with semantically rich entities like people, places, organizations, and ideally relationships such as company acquisitions, instead of, say, Web pages containing such references. For example, tracking and analyzing a politician's public appearances over a decade is much harder than mining frequently used query words or frequently clicked URLs for the last month. The huge size of Web archives adds to the complexity of this daunting task. This paper discusses key challenges, that we intend to take up, which are posed by this kind of longitudinal analytics: time-travel indexing and querying, entity detection and tracking along the time axis, algorithms for advanced analyses and knowledge discovery, and scalability and platform issues.

1. MOTIVATION

Big-data analytics for the *Web of the Future* - Web 2.0 (communities, their behavior, etc.) and Web 3.0 (semantic annotations, linkeddata.org, etc.) - has been a hot topic for some time. However, the *Web of the Past* is an equally important topic for both academics and real-life applications. Academically, longitudinal data analytics is even more challenging and has not received due attention. The sheer size and content of such web archives lends itself to wide applicability for analysts in a great number of different domains.

National libraries and organizations like the Internet Archive (archive.org) and its European sibling (internetmemory.org) have been capturing Web contents over decades. These archives host a wealth of information, providing a gold mine for sociological, political, business, and media analysts. For example, one could track and analyze public statements made by representatives of companies such as Google or Tandem Computers, characterizing the evolution of patterns in their attitude towards energy efficiency. Another example could be tracking, over a long time horizon, a politician's public appearances: which cities has she/he visited, which other politicians or business leaders has she/he met, and so on. Analyses of this kind could also be carried out on large news archives, but this can be seen as variant of Web archive analytics; moreoever, the Web (and especially the recent Web 2.0) has a wider variety of coverage, potentially leading to the discovery of more interesting patterns and trends.

Web archives contain timestamped versions of Web sites over a long-term time horizon. This *longitudinal dimension* opens up great opportunities for analysts. For example, one could compare the notions of "online friends" and "social networks" as of today versus five or ten years back. Similar examples relevant for a business analyst or technology journalist could be about "tablet PC" or "online music". This requires finding all Web pages from certain eras that contain these and/or other related phrases. Unfortunately, this is beyond hope today. Web archives like the Internet Archive provide URL-based access only, via the Wayback Engine. For a given URL, you can retrieve all archived versions of the page; then you can navigate a version on a per-site basis and will be automatically connected to the proper version as of the same snapshot. But this kind of time-travel browsing does not work across sites. There is no support for keyword search along the time axis at all. The NutchWAX open-source software has been tried for archive search, but has not been deployed for public access. Searching for phrases (i.e., multiple keywords occuring contiguously in a page) is computationally much harder (requiring position indexes, etc.), and way beyond today's capabilities. The goal is to allow rich text queries with a temporal filter, such as {"*tablet*

*The authors are the main investigators of the LAWA project, a brand-new project funded by the European Commission, see `http://www.lawa-project.eu` for details.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA..

PC" Europe market} *@ [2003-2007]*, rank the results by time-aware relevance, and aggregate them into a suitable form for subsequent analytics (e.g., into some form of temporally extended text cube).

Analysts are not interested in text or Web pages per se, even if the underlying sources are in text or multimodal form. Instead, they want to see, compare, and understand the behavior of (and trends about) entities like companies, products, politicians, music bands, songs, movies, etc., thus calling for *entity-level analytics* over Web archives. For the tablet-PC search, they would ideally obtain information grouped by named entities like Apple Corp., Microsoft Corp., etc. - combined with the time dimension, for example, by year. Likewise, the matches for "tablet PC" would ideally be based on product names rather than the literal text, to capture also related products such as e-book readers. This calls for lifting the entire archive contents, or at least the slices that are relevant for this analytic task, from the text level to the entity level: detecting named entities, resolving ambiguous names, tracking the same entity in its mentions over extended time periods. Obviously, this is a daunting task, regarding both semantics and scalability, already for current corpora in digital libraries (e.g., PubMed) or enterprises. As entities morph and get renamed over time, e.g., by company acquistions or mergers, this is a grand challenge for large-scale longitudinal analytics.

This paper elaborates on these challenges, identifies specific technical aspects and discusses the problem space. We address semantic and scalability issues. To appreciate the latter, let us merely point out that the Internet Archive currently holds more than 150 Billion versions of Web pages, captured during the timeframe from 1996 until now. Its coverage is getting sparser as Web contents has become so diverse, dynamic, and humongous. A high-coverage archive would have to be an order of magnitude larger.

2. CHALLENGE: TIME-TRAVEL INDEXING

One of the fundamental underpinnings of the envisioned kind of longitudinal analytics is indexing for time-travel queries. The general form of these queries is $t_1 t_2 \ldots t_m @ [T_{low}, T_{high}]$ where the t_i are text terms and T_{low} and T_{high} are the boundaries of a time interval of interest (time points are a special case). In the simplest case, terms are just single keywords, but analysts also need phrases (e.g., product names, campaign slogans, quotations) and may even use additional constructs like negation or distinguishing optional from mandatory terms (e.g., in analyzing intellectual-property issues). A good design for an index to support such queries is not obvious at all. It entails difficult issues regarding 1) the choice of data structures, for example, multidimensional index trees versus IR-style inverted lists versus hash-based synopses, 2) the challenge of efficiently building this huge index (for a given data structure) and incrementally maintaining it, and the associated consistency issues for concurrent analytics tasks, 3) the organization of the index on scale-out platforms so that time-travel queries can be run with high throughput, user-acceptable latency, high availability, and low cost (incl. energy-efficiency). Note that throughput is an issue, even if analysts are a rare species, compared to Facebook users. The reason is that complex analytic tasks may trigger, under the hood, a large number of simpler time-travel queries.

Data structures: The choice is widely open. A database person, at first thought, would most likely advocate a *multidimensional index structure* like an R-tree or perhaps a tailored time-key index like the Multiversion B-tree or the TSB-tree. However, mapping our data space onto one of these structures is full of problems. First, the high dimensionality of the text-term aspect prevents a straightforward mapping; there is no way to support a million text terms by a one-million-dimensional R-tree. And we do need to support multidimensional queries that consist of several terms. Second, the processing of multidimensional range queries over these indexes would result in non-sequential, traversal-style access patterns, thus hurting the effectiveness of the hardware data caches. With many-core processors and flash-based storage, access locality is absolutely crucial for performance. An alternative approach is to adopt the IR paradigm of *inverted lists*. There is usually, one list of postings (document identifiers and associated payload data such as precomputed scores) per term. Each list can be compressed extremely well (see IR literature and also the techniques used by major search engines), and this results in excellent sequential performance. However, with the temporal dimension folded into such an index, an inverted list would contain the postings for all document versions across the entire timespan of the archive. Consequently, lookups for time points or short intervals are penalized by this blow-up in the version space. This problem is aggravated for phrase search. Unless we constrain the flexibility of the analyst by pre-identifying interesting phrases, we need a position index. Here each per-word list contains postings for each occurrence of the word in each document. Blending this kind of index organization with the temporal dimension in an optimized way is a formidable challenge.

Index build: MapReduce is a popular paradigm for building huge indexes on distributed storage. For standard text indexes, this is indeed a great way of harnessing scale-out architectures (and algorithmically not that different from parallel-database techniques). We are studying how to exploit and extend this paradigm in order to build the combined text-time indexes that we need. A key option pertains to the creation of combined text-and-time indices vs separate time and text indices.

Assume each document has a lifespan defined by its last update to current time. At the next update, the lifespan of the document expires and a new one emerges. Thus, the same document, as it evolves through time, is essentially viewed as a series of different documents. In this setting, one could create a MapReduce job (actually a series of MapReduce jobs) that builds text indices, per term, per preset time interval. This is logically equivalent to first creating text indices with posting lists where each posting (docID, term, score) is annotated with a lifespan and then partitioning the posting lists by time intervals. The approach embeds lifespan information with each posting, merging the time and text indices. However, building such an index creates issues such as how to handle documents whose lifespans overlap the indexed time intervals? Replication of relevant posting list entries is a solution. But this exacerbates an already difficult issue: space. An even more formidable task is deciding which should be the preset time intervals for which to build the indices. At one end, large preset time intervals defeat the purpose. At the other, small index time intervals will be inefficient for queries with large time horizons, since many time-interval indices would need to be merged at query time.

A promising solution might be to build several overlapping time-interval indices, at different time-interval granules and employ the best selection of time indices at query time. These time-interval indices can be hierarchically organized, creating in essence an index of indices, which can be traversed to discover the optimal individual indices to use. Deciding this is a difficult optimization task on its own right.

Another approach would be to build separate indices for time and text. The interesting issues here pertain how to best utilize the MapReduce framework to build time-interval structures and which structures lend themselves to better buildup with MapReduce. For more elaborate structures, which are candidates for time indices (such as interval trees and segment trees), it is not clear how to utilize MapReduce for building and efficiently accessing them at query time.

3. CHALLENGE: QUERYING & RANKING

Query processing: With a rich suite of indexes and synopses, the query processor faces many choices in combining the system's data structures. Querying text and time in an integrated manner is a largely unexplored territory. Prior work on news mining typically assumes that timestamps are high-quality metadata. This is not the case at all in Web archives, and this in turn implies that many queries are explorative with wide time-range conditions and complex search conditions about phrases (contiguous words) or soft phrases (nearly contiguous words within a proximity window).

Ranking models: For conventional text search, the ranking of query results is based on word-occurrence statistics, including the idf measure (inverse document frequency) for the specificity of terms. For time-travel queries, the situation is more complicated. A once rare word may now be used in an inflationary manner, so it would now have low weight in the score aggregation for a multi-keyword query. But when a temporal query travels back to that former period, the idf value back then matters. For example, in the query "online friend"@August2002, "friend" would now have low weight but should have high weight as of 2002. Similar issues arise with "static" authority measures such as PageRank, as they are no longer that static anymore in the context of longitudinal archives. Statistical values like idf change continuously with every new update, as they are corpus-dependent (and not confined to a single document). Approximation techniques at lower cost are presumably sufficient, but finding good trade-offs with (probabilistic) guarantees on the deviation error is difficult. Hence, the time dimension affects everything: not just data, but also its indices and their maintenance, and the related statistics and metadata that govern ranking models.

With phrases as query conditions, there are further difficulties. Ideally, we would like to consider the idf value of an entire phrase rather than merely aggregating the scores of the constituting words. But this cannot be precomputed, as it may be only now that we realize the interestingness of a phrase like "online friend" and would now like to pinpoint the onset of this emerging phrase years ago. Finally, bursty-ness in time could be an important ingredient in the ranking of search results. For example, an analyst may look for interesting time points in the longitudinal answers to a query "tablet PC". This should ideally return pages on the 2002 edition of Windows for tablet PCs, the launching of the iPad in 2010, the revival of e-books, and also salient points or periods for more specific results such as intensive press coverage of specific products in certain regions of the world. Here, interesting points could be found by considering the "first derivative" of measures like idf, PageRank, etc.

4. CHALLENGE: ENTITY TRACKING

Entity detection: Detecting named entities in Web pages and thus lifting the entire analytics to a semantic rather than keywords level is a grand challenge already for standard text mining. The difficulties arise from name ambiguities, thus requiring a disambiguation mapping of mentions (noun phrases in the text that can denote one or more entities) onto entities. For example, the mention "Bill Clinton" can be the former US president William Jefferson Clinton, but Wikipedia alone knows five or so other William Clintons. If the text says only "Clinton", the number of choices increases, and phrases like "the US president" or "the president" have a wide variety of potential denotations. For established kinds of data cleaning and text mining, methods for entity resolution (aka. record linkage) have made reasonable progress (e.g. by using statistical learning for collective labeling), and could handle a good fraction of such cases.

Entities in time: In the Web archive case, some additional aspects are assets while others pose major obstacles. The timestamp of an archived Web page can help to narrow down the disambiguation candidates for phrases like "the US president". Similarly, the connection with previous and successive versions of the same page can help to identify changes at specific timepoints, which may in turn be cues for entity resolution. Cases where the temporal dimension introduces new complexity are when names of entities have changed over time. Examples are people's name changes after getting married or divorced (or simply out of some mood), or organizations that undergo restructuring in their identities. Bell Labs is a notorious example; a simpler one is Tandem Computers, a leading company on highly available, scalable systems in the 1980s. Suppose a technology-and-business analyst wants to track companies that used products of Tandem Computers, over the last 30 years (Web archiving does not go back that long, but there are digitized news archives from this era). Tandem was acquired by Compaq, which was later acquired by HP; the NonStop product line (incl. NonStopSQL) has many instances with all kinds of naming variations and still exists today. So we need to identify, from the site captures of Web archives, all mentions of this business entity, its products, and also the enterprises that employed one these products over the years. This would allow us to construct an entire timeline of how the company and its products were doing over several decades: business tracking at the entity-relationship level, automatically inferred from Web history. Such entity tracking should be combinable with filters and aggregations on keywords or phrases. For example, we could restrict the entire analysis to input sources that contain "zero downtime" or "24x7" or "ultra-high availability". We emphasize again that this could be an ad-hoc interest of some analyst; so hardly anything could be precomputed.

A benchmark proposal: Generalizing the example, a conceivable but currently still elusive benchmark could be the following. For all page versions in a Web archive, with 100 billions of files, identify all entities that are known to Wikipedia at some point in the Wikipedia history. That is, map each mention to the Wikipedia article as of the proper timepoint. For example, when Carla Bruni is mentioned on an entertainment site of July 2005, we should map her name to the former model and singer Carla Bruni Tedeschi,

as reflected in Wikipedia as of this time. But the same name seen in August 2009 should be mapped to the French first lady Carla Bruni-Sarkozy (her offical name now). Here different time periods pose different ambiguity challenges. The timestamps of the archived pages are only of partial help, because the page contents can be older; crawl-based dating is not reliable at all. This is a big issue in dealing with locations (e.g., Mumbai vs. Bombay) and organizations (e.g., Bell Labs vs. AT&T Bell Labs vs. Alcatel-Lucent Bell Labs). The challenge lies in the enormous scale and temporal depth, and, of course, the goal of accomplishing this benchmark task with very high precision and recall.

5. CHALLENGE: EFFICIENT ANALYTICS

Interesting phrases and entities: The envisioned system should support a wide spectrum of analytical tasks, spanning the text, entity, and time dimensions. We want to address different tasks of increasing complexity: frequent phrases, interesting phrases, comparative slicing, time pivoting, and entity-entity as well as entity-phrase co-occurrences. Here an interesting phrase could be one that is salient for a particular time period. This could be modeled by information-theoretic measures like relative entropy. Intuitively, a phrase is interesting if it is frequent in the period of interest and infrequent otherwise. An example could be "yes we can" for the period January 2008 through June 2009. Similar analyses should be possible for interesting entities, returning, for example, Deepwater Horizon for the period April through July 2010. This kind of analytics is algorithmically well understood, but carrying it out on a 100-billion-pages archive is a very ambitious goal.

Text-entity-time analytics: Comparative slicing goes beyond the previous stage by trying to identify salient phrases or entities for different subsets of an archive, where the subsets are determined by ad-hoc filters on phrases, entities, and time. For example, we may be interested in a discriminative analysis of public quotations by key people of Google, for the year 2005 versus the year 2010, and perhaps with focus on European Web sites. Here we need to identify temporal slices of the archive, but also select only those page versions that contain mentions of the entity Google, the word "quotation" or some paraphrase for people's statements, and geographic names indicating Europe. The result could be phrases like "do no evil" (the company motto) for 2005 and "nobody was harmed" (Eric Schmidt's reaction to concerns about privacy) for 2010. This line of analyses is a form of co-occurrence mining in the joint space of text, entities, and time. Another analytic task could be time pivoting: for a given entity, find the most interesting timepoints or periods along with a digest of most salient phrases or entities. One can view this as a generalization of tag-cloud timelines. Supporting all this in a truly ad-hoc manner - without any pre-selected phrases or entities and precomputed statistics - is a challenge.

Efficiency: A good part of such tasks could be addressed by MapReduce-based algorithms in a scalable manner. However, these algorithms also need to be efficient in the sense that they use resources - processors, memory consumption, interconnect bandwidth, and energy - in a cost-beneficial manner. If an efficient algorithm can reduce resource consumption, this pays off directly in a lower electricity bill or the ability to run a higher throughput of independent tasks by a larger number of analysts. Thus, we envision a smart combination of scale-out-oriented scanning, hashing, and merging techniques with index lookups, complex execution plans, and statistical approximation methods. The optimization space has an enormous number of degrees of freedom. Here the notion of an execution plan goes way beyond the traditional kind of query execution plan, as it would also involve phrase-mining, entity-resolution, temporal-aggregation, and statistical-computation steps.

6. CHALLENGE: SCALABLE PLATFORM

Figure 1 shows a system architecture, addressing how control and data flow is envisaged within the LAWA project. The base layer consists of storing Web pages and updating the collection with data from new Web crawls. Data at the base layer is processed using MapReduce to produce so-called primary indices (currently .warc files) for the documents in the collection. We envisage a scalable row-store platform (such as HBase) and HDFS as the storage systems for these.

The primary indices contain the essential data, needed to build richer indices, such as text and time-text indices (of the forms discussed earlier), and other statistical structures, such as Bloom Filters and/or histograms needed to estimate join sizes (e.g. of text and time index files), sketches used to estimate cardinalities of sets, etc.

The top layer, the Analytics Engine, is responsible for processing analytics tasks. In turn, these may be expressed as a workflow of complex queries (e.g., involving joins, range selections, top-K operations, etc.) which are handed to the layer underneath, the Complex Query Processing Engine, which in turn may (or may not) utilize (or even build) additional query-specific indices.

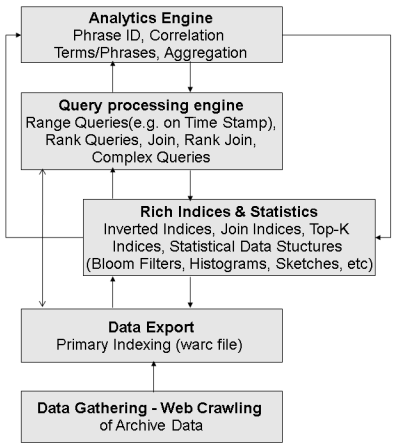


Figure 1: LAWA system architecture

We have discussed some of the key challenges associated with longitudinal analytics over massive data collections of Web archive data. Numerous open research problems are revealed and initial thoughts, trade-offs, and a system organization are presented. As a whole, this area presents new opportunities for our community to design, develop, and deploy solutions, which will help us learn from the past and anticipate the future. It's about time!

Acknowledgements

This work is supported by the 7^{th} Framework IST programme of the European Union through the focused research project (STREP) on Longitudinal Analytics of Web Archive data (LAWA) under contract no. 258105.

Data in the First Mile

Kuang Chen
UC Berkeley
kuangc@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

Tapan S. Parikh
UC Berkeley
parikh@ischool.berkeley.edu

ABSTRACT

In many disadvantaged communities worldwide, local low-resource organizations strive to improve health, education, infrastructure, and economic opportunity. These organizations struggle with becoming *data-driven*, because their communities still live outside of the reach of modern data infrastructure, which is crucial for delivering effective modern services. In this paper, we summarize some of the human, institutional and technical challenges that hinder effective data management in "first mile" communities. These include the difficulty of deploying, cultivating and retaining expertise; oral traditions of knowledge acquisition and exchange; and mismatched incentives between top-down reporting requirements and local information needs. We propose a set of directions, drawing from projects that we have implemented. They include 1) separating the capture of data from its structuring, 2) applying intelligent automation to mitigate human, institutional and infrastructural constraints, and 3) deploying services in cloud infrastructure, opening up further opportunities for human and computational value addition. We illustrate these ideas in action with several projects, including Usher, a system for automatically improving data entry quality based on prior data, and Shreddr, a hosted paper form digitization service. We conclude by suggesting next steps for engaging in data management problems in the first mile.

1. INTRODUCTION

International development organizations aim to improve health, education, governance and economic opportunities for billions of people living in sub-standard and isolated conditions. In many places, this process is becoming increasingly *data-driven*, basing policies and actions on context-specific knowledge about local needs and conditions. Leading development organizations, with the help of research specialists like the Poverty Action Lab, undertake rigorous impact evaluations of development interventions, driven by "belief in the power of scientific evidence to understand what really helps the poor."[1]

Unfortunately, the most under-developed communities are still beyond the reach of modern data infrastructure—in areas with limited power, bandwidth, computing devices, education and purchasing power, among other constraints. Networking researchers often refer to this problem as bridging the "last mile". Even so, as the adoption of mobile phones drive rapidly-expanding network coverage, all but the most remote places seem poised to be connected. While connectivity improves the potential for effective data infrastructure, it alone does not ensure data availability. For database researchers, this last mile is our "first mile" – where essential local data is created, and the hard work of building modern data-pipelines is just beginning.

Our experience working with development organizations around the world has shown that the "first mile" still lacks critical human and institutional capacity for creating modern data-pipelines. [12]. In public health, even basic vital statistics are still not reflected in data-driven processes that affect billions of lives: for example, only 24% of children born in East and Southern Africa are registered [17].

First mile data infrastructure is crucial for delivering effective modern services. Without it, development practitioners, policy makers and communities rely on incomplete, inaccurate and delayed information for making critical decisions. The international public health community warns of an "innovation pile-up": scientific advances, such as new vaccines, will sit idle, awaiting efficient local delivery and adoption [5]. Advances in database technology suffer from a similar innovation pile-up. For want of data, some our best technologies, particularly those in data analytics, are sidelined.

In doing development-minded research, we have observed firsthand that there are many data management challenges that must be addressed to provide for effective data acquisition and interpretation within the first mile. As database researchers, we can provide tools and methods to meet these challenges. However, this requires a shift from our traditional focus on backend infrastructure and algorithms, to the needs of *local data processes* (LDPs) in data capture, quality, throughput and availability in the context of limited human, organizational and technical resources.

In this paper, we first lay out specific data management challenges that we have observed in the field. Next, we discuss promising approaches for addressing these challenges, including concrete examples from our current work. Finally, we suggest some practical next steps for the database community to engage in the first mile.

2. CHALLENGES

In organizations across the developing world, we have witnessed many first mile data challenges. Here we summarize several, with perspective from the first author's work in Tanzania and Uganda with public health and international development organizations.

2.1 Expertise, Training and Turnover

In low-resource organizations, even office-based administrative staff lack expertise in critical areas like database and computer sys-

[1] http://www.povertyactionlab.org

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

tems administration, form design, data entry, usability and process engineering. This is especially true for small grassroots organizations and the local field offices of international organizations, which are assigned the most critical and challenging task of actual service delivery.

It is expensive to provide training and expertise in remote and unappealing locations. For the same reason, it is difficult to recruit and retain high-quality talent. The best staff almost always leave to climb the career ladder; eventually ending up with a job in a major city, or even abroad. Turnover is very high, especially among the young, English-speaking and computer literate. This means that even those organizations that invest heavily in training see limited returns.

2.2 Storytellers versus Structured Data

The field staff of low-resource organizations often have limited formal education. Previous empirical work has shown that uneducated users have difficulty organizing and accessing information in an abstract manner [15]. These characteristics have in turn been associated with a culture of "orality" [11]. According to this theory, oral cultures are characterized by situational rather than abstract logic; preferring nuanced, qualitative narratives to quantitative data. Oral knowledge processes are also aggregative rather than analytic, preferring to assemble complex and potentially conflicting "stories", as opposed to noting down experiences as individual "correct" measurements. Finally, oral communication is usually two-way, with a concrete audience, as opposed to writing, for which the audience can be abstract, temporally and spatially removed, or not exist at all. These characteristics do not translate naturally to field workers capturing structured data using constrained forms destined for a distant, abstract recipient.

2.3 Mismatched Incentives

Like enterprises in the developed world, monitoring organizations are becoming increasingly data-driven. Indeed, the World Bank reports, "Prioritizing for monitoring and evaluation (M&E) has become a mantra that is widely accepted by governments and donors alike." [10]. On-the-ground organizations face, on one hand, growing data collection requirements and, on the other, the mandate to minimize "administration" overhead, the budget that sustains data management. Some international funders assume that if an on-the-ground organization is effective and worthy of their help, then their reporting requirements can be met with data already being collected [6]. This is wishful thinking and far from reality. Organizations in the local communities are often several rungs down on the sub-contracting or delegation ladder, and are disconnected from the rationale behind reporting requirements. Ironically, local organizations create one-off, haphazard, heavily tailored "point solutions" that minimally meet essential reporting requirements, often at the expense of local information needs. The notion of data independence is painfully missing, leading to processes that are inefficient, inflexible to change, and hard to staff.

In one large urban Tanzanian health clinic, we observed that patient visit data was recorded by hand twice and digitally entered twice. Staff wrote by hand first, in a paper filing register, which the clinic used for day to day operations, and next, on a slightly different carbon-copy form. The first copy, for the local ministry of health, was digitally entered onsite; the second copy, for an American funder, was shipped to headquarters and entered there.

Another misaligned incentive is that generating clean, aggregated, long-term data (months to years) that is useful for top-down evaluation and policy is very different from generating the more nuanced, individual, short-term data useful for decision making at the local level. For example, a funder may be interested in a quarterly count of patients with malaria, while a health clinic wants to know which malaria patients from yesterday require follow-up. In emphasizing the former, the latter is often ignored. In the example from Tanzania described above, the local health clinic had no access to digitized records, despite onsite data entry. They could only rely on searching through paper forms. In a busy, resource-constrained environment, this means that patient records were often not referenced during treatment. In turn, this lack of direct benefit creates no incentives for local practitioners to generate quality data consistently. Finally, reporting to funders means emphasizing one's successes, while improving operations often requires learning from your own mistakes. This subtle bias suggests that the most important insights from the data probably do not surface.

3. EMERGING DIRECTIONS

In this section, we propose some technical directions for addressing the challenges listed above. The general approach is to better segment the data workflow, and to either automate certain high-skill tasks, or to delegate work in ways that better suit the incentives and capabilities that are available.

3.1 Separate Capture from Structure

First of all, we believe it is important to distinguish between data *capturing* and data *structuring* tasks. The first refers to extracting some bit of information or knowledge from the real world and recording it in a persistent form. The second refers to organizing, categorizing and quantifying this information, often according to some pre-ordained structure. Our experience suggests that front-line field workers are the best suited to capturing important local information, due to their local contextual knowledge, familiarity with the community and in some cases, oral culture. On the other hand, structuring tasks require more literacy, training and knowledge about increasingly specific data vocabularies and schemas. The goal should be to move structuring tasks to where the incentives and capabilities are most appropriate.

This suggests a number of directions for future research. One project, Shreddr, described in further detail in the next section, allows field workers to capture information using existing and familiar paper forms. These forms are iteratively digitized, using a combination of automated and human-assisted techniques. Another project, Avaaj Otalo, is extracting important statistics about farm cultivation, pest infestation and mitigation directly from farmers' own recorded questions and answers [14]. Increasingly affordable technologies like GPS-enabled camera phones or digital paper[2] suggest even more powerful possibilities [9].

In general, these techniques trade off more contextually appropriate input techniques, for more uncertainty in the initial results. Capture by field agents is only the first step in a multi-stage "entropy-reduction" or "denoising" process. Down the data-pipeline, we can interleave a sequence of automated and human-assisted steps to progressively reduce noise, generating increasingly accurate statistics for decision makers, leaving intermediate results explicitly available for local analysis.

3.2 Intelligent Automation

By applying automated techniques such as optical-character recognition, voice recognition and statistical prediction, we can reduce local expertise and training requirements. For example, intelligent prediction can be used to simplify data entry. Instead of requiring a user to type in a field value, we could ask whether the most probable value is accurate. The approach of converting entry into validation can improve efficiency and quality, and can potentially even remove the requirement for keyboards and computers in some settings.

We can also use statistical techniques to more effectively organize tasks, including automatically deriving form designs based

[2]http://www.anoto.com

on prior data, including appropriate field constraints. The Usher project, described in the following section, applies these and other techniques to improve the quality of entered data. Essentially, these adaptive techniques learn from existing data, applying the results to mitigate the lack of management, formal processes, staff expertise and high turnover that can stifle other forms of organizational learning.

3.3 Leverage the Cloud and Crowd

Separating capture from structure also allows us to host more of the structuring activities directly in "the cloud", further reducing local data management requirements, and creating opportunities for more intermediaries to provide value. Both the Shreddr and Avaaj Otalo systems are positioned to be hosted services, allowing local organizations to focus on capturing paper scans and audio recordings, respectively, while the structuring tasks are distributed across the Internet. Workers can include staff at headquarters who often have the most direct incentives and motivation to obtaining timely, high-quality data. Moreover, given that many of these projects are directly in support of social goals, we may even be able to rely on "cognitive surplus" in the developed world, in the form of crowd-sourced workers working for social or other incentives [16].

In general, aggregating many "little-data" processes into a smaller number of more traditional big-data activities achieves economies of scale, and better facilitates a variety of value-adding services, including: (1) automatic value estimation, such as OCR/OMR; (2) incremental and elastic scaling of workers with crowd-sourcing; (3) dynamic task assignment according to workers' skills and incentives; (4) reporting and analytics for multiple recipients, including returning data back to the first mile for local usage.

3.4 Bottom-up Optimization

The above discussed directions on optimizing data-pipelines will make timely data available to on-the-ground staff. If we develop contextually appropriate tools that allow these users to perform analysis, we can help organizations self-optimize quantitatively.

In a rural Ugandan village, we developed a simple Excel tool allowing clinicians to view data visualizations of health trends in their community. The key idea was leveraging "found data" from the intermediate results of fulfilling external data collection requirements. The tool was simple: an Excel workbook with macros that tapped into existing data collected from community health workers (CHWs) reports. We created a workbook tab of visualizations featuring PivotCharts like "Patients under 5 years old with malaria by village.", and taught the village doctor in charge of CHWs to create his own PivotCharts. The village doctor delighted in his new-found ability to monitor CHWs through visualizations. We saw that the ability to see and benefit from CHW collected data immediately improved incentives and feedback loops for CHW data collection. Motivated by this simple tool's adoption, we have proposed, as future work, a framework for automatically generating and identifying visualizations that contain "actionable anomalies" [2].

This type of *bottom-up* analytics can improve local decision-making. It allows practitioners to surface locally-important outliers and trends, which may otherwise get lost in higher levels of aggregation. As well, it has the additional benefit of aligning mismatched incentives between local and oversight organizations. Encouraging local data-consumption feeds a virtuous cycle: growing data usage increases the desire to collect higher quality and lower latency data, which then makes the data more useful, and so on.

4. CURRENT WORK

In this section, we describe two of our projects aimed at improving the quality, efficiency and utility of data entry from paper in public health organizations in sub-Saharan Africa. We have found that batch data entry is a key choke-point for such organizations in the first mile, and an early opportunity to improve efficiency, to catch (and correct) errors in the data-pipeline, and to directly and immediately apply lessons learned.

4.1 Usher

Usher is a tool for automatically improving the accuracy of data entry interfaces. The survey design literature provides a number of existing best practices for form design [8]. However, most of these are still heuristics, and implementing them in any given context is still more of an art than a science. Drawing from these best practices, and an information-theoretic entropy reduction model of data entry, Usher seeks to automatically generate a form layout and digital data entry interface that can maximize information gain, input efficiency, and accuracy, for any arbitrary form and dataset.

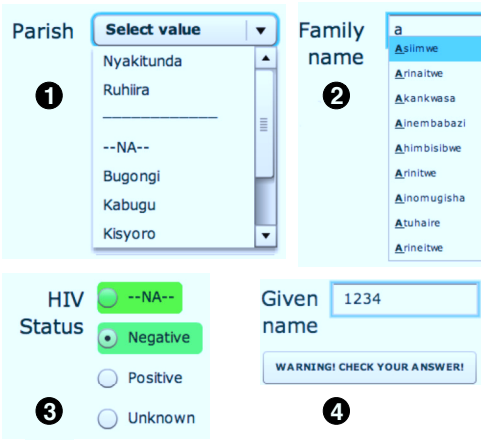


Figure 1: (1) drop down split-menu promotes the most likely items (2) text field ranks autocomplete suggestions by likelihood (3) radio buttons highlights promote most likely labels (4) warning message appears when an answer is an outlier.

Usher is driven by a probabilistic model of relationships between a form's questions and values derived from prior data. Leveraging this predictive ability, Usher provides algorithms for: (1) *reordering* the sequence of form questions to maximize information gain at every point in data entry, allowing for better prediction of remaining fields – similar to what a good form designer might do, (2) *re-formulating the presentation of questions* to make it easier to select more likely choices, and more difficult to select less likely ones (Figure 1 shows a subset of Usher-powered feedback mechanisms that we tested with users), and (3) *re-asking* questions that are likely to be wrong – approximating double-entry (the practice of having two data clerks enter the same form and comparing), but only for values likely to be incorrect, and thus at a fraction of the cost.

Our user experiments working with real datasets and real data entry clerks in rural Uganda demonstrated that Usher can significantly improve input efficiency and accuracy [3, 4].

4.2 Shreddr

Our more recent work on Shreddr takes a "column-oriented" view of data entry, with the hypothesis that automatic decomposition and information theoretic redistribution of data entry tasks, along with novel entry interfaces, can provide significant gains in data entry efficiency. Shreddr works as follows: (1) *extract schema and physical locations* of schema elements semi-automatically from a scanned form, via a simple web interface. (2) *align and shred* images of completed forms into image fragments according to physical locations, and estimate field values via optical character and

mark recognition (OCR/OMR) and Usher. (3) *dynamically re-batch* image fragments by data type and value estimate into worker tasks, and present with Usher interfaces. (4) *crowd-source or in-source* tasks to workers in an elastic labor pool (such as Amazon's Mechanical Turk), or an organization's own workers.

The Shreddr approach to data entry has several advantages. It enables low-fidelity automation to greatly simplify a large percentage of tasks. Since confirmation is often much less difficult for humans than *de novo* entry, it focuses the limited attention of human workers directly on entering the most difficult to guess values. As well, the freedom to order tasks in a "column-oriented" fashion allows control of latency and quality at field-by-field granularity. This means time-sensitive fields can be given priority, and important fields can be confirmed and re-confirmed.

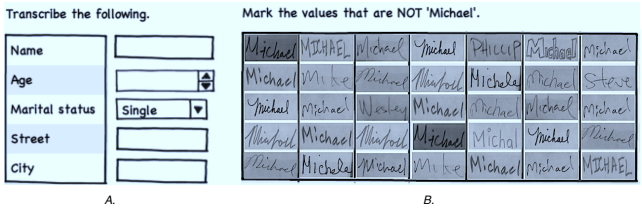


Figure 2: Two example interfaces for data entry.

Columnar-orderings enable several mechanisms for better efficiency. First, workers can better retain mental focus by transcribing similar values, without switching question context—for example, a sequence of only "firstname" values. Second, a column can be sorted by its predicted value, allowing workers to verify sequences with roughly a single value, like "Michael". We can provide user interfaces that essentially allows batch confirmation of several values at one time. We put these techniques together in Figure 2: interface A is traditional row-order entry; interface B is column-ordered validation of sorted "firstname" values predicted to be "Michael". We suspect that batch entry of pseudo-sorted sequences will yield much higher digitization throughput, much like run length encoding in a compressed database column.

5. GETTING STARTED

As we have illustrated, there are a number of first mile data challenges that can be directly addressed by re-organizing and optimizing the local data infrastructure. We believe the database community is well-positioned to make significant contributions in this area. However, to do so, we must recognize some of the implicit assumptions in current database research. We list some of these below in the hope of stimulating discussion that can advance notions about our field, of Computer Science in general, and its applicability to a number of important real-world contexts.

The notion of "too much data": William Gibson observed that "The future is already here, it is just unevenly distributed" [7]. This insight applies to data as well. While we in the database community often talk about the data deluge occurring in the developed world, there is, ironically, far too little data available about conditions in the developing world – data that is relevant to some of the most important challenges and opportunities of the 21st century. While we are very comfortable with issues like scale and privacy in data-rich environments, we are less familiar with circumstances where even the most basic improvements in data availability can enable significant progress in meeting local needs.

The infatuation with "big data": Database researchers take more interest in problems that center on large data volumes. But, because low-resource organizations use tools like Microsoft Access, they tend to fly under our radar. However, their multitude of "little data", each different by culture and environment, also presents an interesting scale problem: the challenge of wide-scale in *contextual diversity*, rather than large-scale in volume.

The myth of expertise: We often assume that competent staff is on hand to implement, administer and use our systems. This thinking is reasonable for many office-based, developed world environments, but if we want to extend the reach of our systems to more people and organizations, we must go further in terms of making our solutions more appropriate for a broader range of skill levels and familiarity with technology.

The most direct route to engaging with global problems is pragmatic. As several early researchers in this emerging field have highlighted, there is a simple formula for achieving success [1, 13]: go to the field, find a good partner organization, and solve their real problems in a empirically demonstrable and hopefully broadly generalizable way. This path leads to interesting and unexpected solutions, including some we may never have thought of otherwise.

6. REFERENCES

[1] E. A. Brewer. VLDB Keynote Address: Technology for Developing Regions, 2007.

[2] K. Chen, E. Brunskill, J. Dick, and P. Dhadialla. Learning to Identify Locally Actionable Health Anomalies. In *Proc. AAAI Spring Symposium on Artificial Intelligence for Development*, 2010.

[3] K. Chen, H. Chen, N. Conway, T. S. Parikh, and J. M. Hellerstein. Usher: Improving data quality with dynamic forms. In *Proc. ICDE*, 2010.

[4] K. Chen, T. Parikh, and J. M. Hellerstein. Designing adaptive feedback for improving data entry accuracy. In *Proc. UIST*, 2010.

[5] C. J. Elias. Can we ensure health is within reach for everyone? *The Lancet*, 368:S40–S41, 2006.

[6] B. Gates. ICTD Keynote Address, 2009.

[7] W. Gibson. The science in science fiction. *Talk of the Nation*, 1999.

[8] R. M. Groves, F. J. Fowler, M. P. Couper, J. M. Lepkowski, E. Singer, and R. Tourangeau. *Survey Methodology*. Wiley-Interscience, 2004.

[9] C. Hartung, Y. Anokwa, W. Brunette, A. Lerer, C. Tseng, and G. Borriello. Open data kit: Building information services for developing regions. In *Proc. ICTD*, 2010.

[10] I. E. G. (IEG). *Monitoring and Evaluation: Some Tools, Methods and Approaches*. World Bank, Washington, DC, 2004.

[11] W. J. Ong. *Orality and Literacy: The Technologizing of the Word*. Routledge, 2002.

[12] T. S. Parikh. Engineering rural development. *Commun. ACM*, 52(1):54–63, 2009.

[13] T. S. Parikh, K. Ghosh, and A. Chavan. Design studies for a financial management system for micro-credit groups in rural India. In *Proc. Conference on Universal Usability*, 2003.

[14] N. Patel, D. Chittamuru, A. Jain, P. Dave, and T. S. Parikh. Avaaj otalo: a field study of an interactive voice forum for small farmers in rural india. In *Proc. SIGCHI*, 2010.

[15] S. Scribner and M. Cole. *The Psychology of Literacy*. Harvard University Press, 1981.

[16] C. Shirky. *Cognitive Surplus: Creativity and Generosity in a Connected Age*. Penguin, 2010.

[17] UNICEF. The state of the world's children 2008: child survival, 2008.

Managing Structured Collections of Community Data

Wolfgang Gatterbauer
University of Washington
gatter@cs.washington.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

Data management is becoming increasingly social. We observe a new form of information in such collaborative scenarios, where users contribute and reuse information, which resides neither in the base data nor in the schema information. This "superimposed structure" derives partly from interaction within the community, and partly from the recombination of existing data. We argue that this triad of data, schema, and higher-order structure requires new data abstractions that – at the same time – must efficiently scale to very large community databases. In addition, data generated by the community exposes four characteristics that make scalability especially difficult: (*i*) *inconsistency*, as different users or applications have or require partially overlapping and contradicting views; (*ii*) *non-monotonicity*, as new information may be able to revoke previous information already built upon; (*iii*) *uncertainty*, as both user intent and rankings are generally uncertain; and (*iv*) *provenance*, as content contributors want to track their data, and "content re-users" evaluate their trust. We show promising scalable solutions to two of these problems, and illustrate the general data management challenges with a seemingly simple example from community e-learning ("ce-learning").

1. A VISION: MASSIVE COMMUNITY E-LEARNING WITH PAIRSPACE

We will argue that management of *collections of community data* requires a new abstraction that does not fit well in the common dichotomy of data and schema information. We illustrate this idea with the vision of a massive online question-answer learning community of users, grouped around a hypothetical tool we refer to as PairSpace. We prefer to keep the overall setup simple. This is a concrete community data management scenario that illustrates the main issues in this paper, while at the same time, seems to have a simple relational implementation. Note that the underlying challenges naturally extend to more complex and general *community content management scenarios*.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

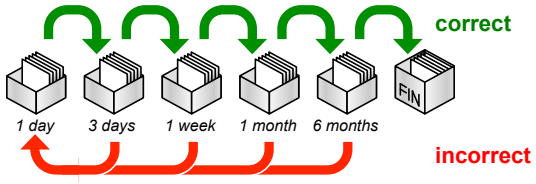


Figure 1: Spaced repetition with flashcard learning: Repetition intervals increase for subsequent boxes.

PairSpace is a huge shared repository of learning nuggets organized into question-answer (Q&A) pairs that combines (*a*) flashcard learning with (*b*) spaced repetition and (*c*) a community built around it. *Flashcards* are sets of cards with a question on one side and an answer on the other. These cards are used as a learning drill to aid memorization of learning material through what is called "active recall[1]:" given a question, one produces the answer. Furthermore, those Q&A pairs are usually grouped into *collections* of a similar nature, i.e. meaningful learning units. Examples are the vocabularies of one lesson in a high-school book, or the standardized questions to pass the US driving license in the State of Washington. Almost any cognitive subject can be translated into such a Q&A format[2].

Spaced repetition is a learning technique with increasing intervals of time between subsequent reviews of learned material. Items to memorize are entered into PairSpace as Q&A pairs (virtual flashcards). When a pair is due to be reviewed, the question is displayed, the user attempts to answer the question, and – after seeing the answer – decides whether he answered it correctly or not. If he succeeds, then the pair gets sent to the next box, if he fails it gets sent back to the first box. Each subsequent box has a longer period of time before pairs are revisited (Fig. 1)[3]. Imagine a daily

[1] In active recall, pieces of information are *actively* retrieved from memory as opposed to *passive review*. See [10] for a recent discussion.

[2] Further examples are: general cultural facts (such as world countries and their capitals), competition results for sports fans (e.g., Who won the 2010 World Cup?), film facts for movie buffs (e.g., Who played William of Baskerville in "The name of the rose" of 1986?), often asked terms during GRE and their synonyms, important paragraphs or cases in law, details on the periodic table in chemistry, multiplication tables in mathematics, names of bones and their location in the human body for medical students, basic formulae in any science, or lists of common abbreviations in computer science (e.g., What does MVD stand for?).

[3] The idea of spaced repetition traces back to the early 1930s, but only became later widely know as Pimsleur's graduated-interval recall [15], or "Leitner system", or "Ebbinghaus Forgetting Curve". While not widely popular in the USA, flashcard learning is hugely popular in Europe with several hundred, mostly offline flashcard programs

morning routine in which a user repeats the learning nuggets that are due that day as suggested by the system.

The third aspect is that those collections of pairs can be shared, re-used, and even re-combined. We envision *one centralized and massive repository* of Q&A pairs for all disciplines, languages and kinds of human knowledge. This is the one central place (with obvious positive externalities) where learners go for repetitive learning needs to find relevant collections of information nuggets, to upload or combine pairs into new collections, and to train regularly. Major value of the stored information lies not just in the individual Q&A pairs (e.g., the translation of English "go" into Spanish "ir" can be easily found in any free online dictionary), but in the *collection of these information nuggets into meaningful units of information* whose mastery together allow the learner to acquire a certain skill level (e.g., passing the knowledge-based driving test). And this value is important to leverage when helping users find the right pairs, collections, or even other users with similar interests.

EXAMPLE 1 (PAIRSPACE SCENARIO). *Alice is learning Spanish. She uplods Q&A pairs of her first lesson. Bob is learning Spanish too and discovers Alice's Spanish 1 lesson in* PAIRSPACE. *His girlfriend is Mexican and has taught him to use* `andar` *instead of* `ir` *for* `go`. *He changes his Q&A pair* (`go`, `ir`) *to* (`go`, `andar`). *Next assume Charlie is searching for basic Spanish lessons. What should the system return to Charlie, and how should it present this result?*

2. CHALLENGES FOR MANAGING COLLECTIONS OF COMMUNITY DATA

The simple scenario of Example 1 already poses several challenges of how to search, return and present the results.

• *What to return?* Should the system return the collection `Spanish 1` of either Alice or Bob? Should it present them as a derivation of each other with Bob's collection as the most recent, or Alice's as the original? Should it return just the intersection of Alice's and Bob's collections as new *derived* collection? Should it present and mark the tuples (`go`, `ir`) and (`go`, `andar`) as possibly *conflicting* pairs? Stated more abstractly, given two or more collections as input, how to inform and return to the user the *structural variation in collections*? How can the system learn to *suggest* new derived collections that fit the purpose of the user?

• *How to bundle and present* the results to the user? Can we take advantage of new "return structures" imposed by collections instead of returning individual pairs in an all-too-familiar list-based fashion (cf. discussion in [3])? If we have several partially overlapping and often complementary or contradicting collections, should we return collections or tuples by majority or by diversity (cf. [20])? Should we cluster these collection into *meta-collections* in the search results, i.e. go one level further in the abstraction?

• *How to search?* How does the user specify the information she is looking for, i.e. what is the appropriate search paradigm for query formulation? What is the appropriate query language that – though possibly hidden from the user – allows to express the user's search needs?

found on the Web. For example, Phase-6 is a German company entirely built around an *offline* flashcard learning software that is used at 3.000 German schools and has supposedly been sold more than 500.000 times (source: `http://www.phase-6.com`). The fact that there are hundreds of software tools available supports the thesis that *one unique, and online* PAIRSPACE would a valuable tool to users, but nobody has yet figured out the perfect solution to bring this to *massive* dimensions (cp. Friendster and LinkedIn before Facebook).

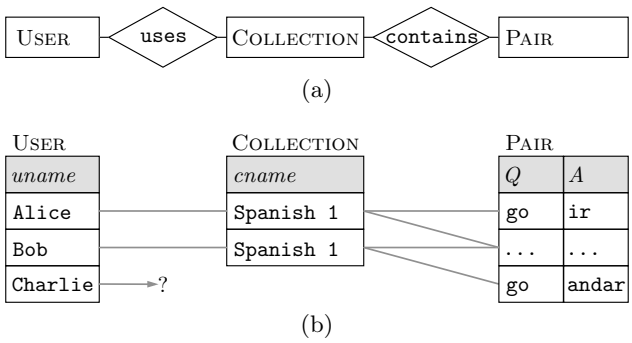


Figure 2: An attempt at a relational encoding of PAIRSPACE: (a) an ER model, and (b) an instance.

• *What to include in ranking?* What are those explicit or implicit features or associations that can be leveraged to learn and return *relevant* information to the user? Obvious candidates are the following: (*a*) *Semantic or syntactic similarity*: How can one address synonymy and polysemy? For example, the expression `bank` can represent "river bank" in English, "bench" in German, and a "financial institution" in both languages. (*b*) *Structure*: What is the generally appropriate way to think about the relative importance between pairs or items and collection of items? (*c*) *Trust or reputation*: What is the appropriate abstraction of trust between users in this scenario? How can different levels of trust be combined, i.e. should they be defined either in a vote-based (democratic, weight-based) manner, or rather a rule-based (strict, preference based) manner. For example, Charlie may specify to trust his school teacher strictly more than any other people. In such preference or rule-based scenarios, the actual value of the weights does not matter, but rather the partial order between preference relations [18]. (*d*) *Provenance*: What kinds of provenance are suggested by this scenario, such as "social provenance" [3], or derivative provenance? Should identical pairs specified by different users be linked to each other? How can one define provenance on collections of items? How to incorporate all those forms of provenance into an appropriate ranking function? How to support querying those combined forms of provenance, e.g. to support *explanatory queries* over this repository?

One fundamental problem is already the question of how to best *store, manipulate, and update* all involved information over time. Figure 2a shows a simple ER model of the relation between users, collections and pairs in our scenario. Figure 2b is a simplified depiction of the scenario of Example 1. If the collection of Alice contains 100 tuples, then storing Bob's *incremental variation* would take $< \frac{2}{100}$th of the space of Alice's original lesson. For the sake of discussion, let's call replicating all pairs as the *explicit representation* (alternatively eager or materialized), and some other representation that stores only the difference as *implicit* (alternatively lazy or virtual). But space is not the major issue here: even if the explicit representation is stored in a compressed form, valuable information about the *relative derivation* or evolution of content is lost. Note, that during querying, value lies not so much in the individual pairs or collections, but rather in the knowledge of how close two or more *collections relate to each other*. In turn, storing only the implicit

information may decrease the access to the actual data considerably. Hence, there is this inherent trade-off between having the data explicit, or the relative derivations explicit. How to update those derivations if content evolves and users add, update, delete or transfer pairs between collections?

Summing up the three main challenges that this seemingly simple scenario of managing three kinds of entities (items, collections, users) in a community scenario raises are: (1) What is the right abstraction for the logical and physical representations of this partly redundant, partly overlapping information, grouped into vastly overlapping bundles? (2) What is the right abstraction of a data manipulation and query language that allows one to reason in terms of collections rather than items? (3) How to evaluate relative importance over the triple concepts of (items, collections, users) in a sound and principled way? In addition, how to reason about (*i*) inconsistency, (*ii*) non-monotonicity, (*iii*) uncertainty, and (*iv*) provenance at the level of collections?

3. WHY EXISTING MODELS AND APPROACHES DON'T SUFFICE

Here we briefly summarize related work that focuses on these challenges but fails short in solving them entirely.

The overall area falls into what is classified as *sharing systems* in [5] where users together build shared structured knowledge bases or a consistent data synthesis. In our scenario, the users do not share the goal of structured knowledge creation, but rather want to find individually fitting collections of Q&A pairs that fit their respective learning needs. Our scenario is clearly different from *data integration* or *data fusion* where the goal is to create *one* unified view on the data [12]. Instead, we want to efficiently manage, find, and compose meaningful collections/bundles/structures of base data that evolve over time. Our challenges are also reminiscent to those of dataspaces [11], where the focus is on incremental ("pay-as-you go") integration. The value of the system increases over time with the number of matches between the data. In our scenario, collections of items have different meanings to different users at different times and need to be managed from day one.

The scenario is also related to the problem of *conflict resolution in community databases* with the goal of automatically assigning each user in the system a unique value to each key [18, 9]. However, in our scenario, content import is "pull" instead of "push," i.e. users actively search for content. In the scenario of searching over Yahoo! answers [1, 2], the goal is to order the set of question-answer pairs according to their relevance to the query. In our scenario, the goal not just to rank just pairs, i.e. user generated content, but rather (or alternatively) *collections* of pairs (which may exist or may be re-combined), or to suggest relevant users.

Our notion of collections is also very reminiscent of *superimposed information* [14], i.e. data that is placed over existing information to help organize and reuse items in these sources. One main difference to superimposed information management is the community aspect: we have different alternative and overlapping collections of base information, and value lies not just in the groupings, but also the *difference between alternative groupings*. The concept of finding and managing associations on top of base data is also related to *inductive databases* [16] and *pattern-base management systems* [4]. Both try to manage rules built upon base data as separate information inside an enhanced DBMS. The difference is that the collections that we are interested in do not have in general an *intensional semantics*. That means they cannot *by themselves* be expressed in a short implicit form, e.g., by a query (cf. [17]). Rather, we are interested in the *incremental and evolving differences* between collections of base data. And we want to leverage this information about "data interference" during the ranking process. Our scenario is also reminiscent of revision control systems (RCS), such as SVN, which manage incremental changes to documents, programs, and other information, and optionally include compression. But while RCSs can store differences efficiently, they do not expose general query facilities to search for meaningful differences, which is an essential ingredient in search in community databases.

4. A NEW HOPE

We propose two complementary approaches to deal with a subset of these challenges.

(1) Rule-based, non-monotone preference relations. An important feature of a community database is providing ways to establish, measure, and show fame/trust/reputation of content and users. This is especially important for a system that needs to provide the users functionality of *revoking* previously stated information ("non-monotone reasoning"). If the user contributions seep deep into the system and other users build upon it, then undoing can be very difficult [5]. Take as an example the spread of wrong information in a social network, where people trust people by a *transitive closure of trust* (people trust people who trust people etc.).

The approach that we promote here is exchanging *vote-based* trust systems (or at least enhancing them) with a *rule-based reputation system* built on *conditional trust*. Conditional trust relations are basically trust mappings at a data-instance level. Trust mapping is a preference-based inference rule or a default statement that a user is willing to accept another user's data value in the absence of their own values. Priorities are further used to specify how to resolve conflicts between data values coming from different trusted users [18]. Conditional trust now allows users to specify selected attributes and thus takes this trust mappings from a schema to a data focus, quite similar to conditional functional dependencies (CFDs) [6] further specifying standard FDs. It is not trivial to reason efficiently in such a rule-based system with the transitive closure of trust, and in the presence of cycles. However, we have shown in very recent work [9] that a unique semantics can be defined that allows to calculate revokes, that means changes in the data instances, efficiently (linear in size of the data and quadratic in the size of the network even in the presence of cycles).

We see two interesting questions with this approach: (*i*) When porting such a rule-based semantics into a richer social context and from the schema-level to data-level, can one still guarantee efficient scalability in both the size of the shared data and the size of the trust relations? (*ii*) Can we raise the level of abstraction for rules further: When treating preference rules as data, can preference-rules on preference-rules still allow an efficient semantics? In recent work [7], we have shown that reasoning in modal logics (arbitrary nesting of annotations on annotations) can be encoded efficiently on top of the relational model. Can we still find efficient encodings of such higher-order rules that can be managed on top of the relational model?

(2) Ranking collections of data in an efficient probabilistic framework. We mentioned before the problem of ranking results relevant to the user, which is also studied in work on incorporating social media into learning-based ranking methods (e.g. [2, 3]). Such ranking methods need to take into account *uncertainty* at several levels: uncertainty about the semantic relevance of the results to the user's query, uncertainty about the user's search context, uncertainty of mapping of keywords to various schema or data information. Furthermore, the ranking method needs to allow for structured queries (e.g. "Find collections on Spanish with the word `go` in one pair and used by trusted users").

A natural way to deal with structured queries over uncertain data is to interpret all uncertainties or weights in the system as probabilistic values in the interval [0,1], and then apply a structured query paradigm that ranks results by their relevance as common in probabilistic databases (PDBs). Recent work on PDBs has shown that the parameters of the ranking functions can be learned from user preferences [13]. Thus, treating the problem of query answering as those of query answering over PDBs allows one to learn the values, then to support complex querying and decision-making over data. The main problem with this approach is its computational complexity: evaluating conjunctive queries over tuple-independent PDBs is well known to be already #P hard, in general. For example, complexity-wise, the above query over the schema from Fig. 2 would correspond to evaluating a Boolean conjunctive chain query $q :- R(x), S(x, y), T(y)$ which is #P hard in our case (many-to-many relations between all entities in Fig. 2a).

The solution that we advocate is our recently introduced technique of *query dissociation* [8]. The basic idea is to slightly change the semantics of probabilistic query evaluation, and to then calculate the ranking score of result tuples with a few materialized views and a single query plan that guarantees efficient evaluation for *every* conjunctive query. In theory, this approach replaces a #P hard problem with a PTIME algorithm. In practice, it allows *existing DBMSs* to evaluate probabilistic conjunctive queries over data instances that have been infeasible for previous approaches.

5. FINAL THOUGHTS

We have pointed to the challenge of organizing and managing *collections of community data* with the example of PairSpace, a fictitious centralized and massive repository for Q&A learning. We envision this shared space as a nucleus that can grow as *the* single repository for general e-learning or even knowledge repositories with more complicated structural knowledge than Q&A pairs (cf. Wikipedia info boxes as nucleus for structured knowledge extraction from the Web). At the same time, solving this structurally simple data management problem will help shape our thoughts of how to approach more complicated structural collections of human knowledge, such as general *community content management systems*, or even those that do not have a naive implementation in the relational model. The most timely such challenge is managing the variance of the human genetic population.

The 1000 Genomes Project (see `http://1000genomes.org`) creates a collection of 1000 human genomes and aims at achieving a complete representation of the structural variation (inserts, deletions, inversions, translocations [19]) in the human genome. Now imagine a massive database that allows to query the human genetic variation of 1 billion genomes, and to correlate these variations with variations in clinical data. This massive data management system clearly derives value from efficient handling of higher-order structure that is imposed on top of the base genetic information (variations between structured collections of base pairs), but also variations in the variations themselves (higher-order structural variations). Effective solutions to biologists for this kind of problem have not yet been found by the data management community.

We strongly believe that investigating the right data model for a massive PairSpace will at the same time help the global learning community, and will lead to better understanding of appropriate abstractions for managing more general massive collections of community data.

Acknowledgement. We like to thank Phil Bernstein and Alexandra Meliou for helpful comments and discussions. This work was supported in part by NSF grant IIS-0915054 (the BeliefDB project: `http://db.cs.washington.edu/beliefDB/`).

6. REFERENCES

[1] E. Agichtein, C. Castillo, D. Donato, A. Gionis, and G. Mishne. Finding high-quality content in social media. In *WSDM*, pp. 183–194, 2008.
[2] E. Agichtein, E. Gabrilovich, and H. Zha. The social future of web search: Modeling, exploiting, and searching collaboratively generated content. *IEEE Data Eng. Bull.*, 32(2):52–61, 2009.
[3] S. Amer-Yahia, L. V. S. Lakshmanan, and C. Yu. Socialscope: Enabling information discovery on social content sites. In *CIDR*, 2009.
[4] I. Bartolini, P. Ciaccia, I. Ntoutsi, M. Patella, and Y. Theodoridis. The PANDA framework for comparing patterns. *Data Knowl. Eng.*, 68(2):244–260, 2009.
[5] A. Doan, R. Ramakrishnan, and A. Halevy. Mass collaboration systems on the world wide web. *CACM*, 2010. (to appear).
[6] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), 2008.
[7] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe it or not: Adding belief annotations to databases. *PVLDB*, 2(1):1–12, 2009. (CoRR abs/0912.5241).
[8] W. Gatterbauer, A. K. Jha, and D. Suciu. Dissociation and propagation for efficient query evaluation over probabilistic databases. In *MUD*, 2010.
[9] W. Gatterbauer and D. Suciu. Data conflict resolution using trust mappings. In *SIGMOD*, pp. 219–230, 2010.
[10] D. Glenn. Close the book. Recall. Write it down. *Chronicle of Higher Education*, 55(34):A1, 2009.
[11] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of dataspace systems. In *PODS*, pp. 1–9, 2006.
[12] A. Y. Halevy, A. Rajaraman, and J. J. Ordille. Data integration: The teenage years. In *VLDB*, pp. 9–16, 2006.
[13] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1):502–513, 2009.
[14] D. Maier and L. M. L. Delcambre. Superimposed information for the internet. In *WebDB*, pp. 1–9, 1999.
[15] P. Pimsleur. A memory schedule. *The Modern Language Journal*, 51(2):pp. 73–75, 1967.
[16] L. D. Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, 2002.
[17] D. Srivastava and Y. Velegrakis. Intensional associations between data and metadata. In *SIGMOD*, pp. 401–412, 2007.
[18] N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, pp. 13–24, 2006.
[19] E. Tuzun, A. J. Sharp, J. A. Bailey, R. Kaul, V. A. Morrison, L. M. Pertz, E. Haugen, H. Hayden, D. Albertson, D. Pinkel, M. V. Olson, and E. E. Eichler. Fine-scale structural variation of the human genome. *Nat Genet*, 37(7):727–732, 2005 Jul.
[20] C. Yu, L. V. S. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: diversification in recommender systems. In *EDBT*, pp. 368–378, 2009.

Crowdsourced Databases: Query Processing with People

Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, Robert C. Miller
MIT CSAIL
{marcua, sirrice, karger, madden, rcm}@csail.mit.edu

ABSTRACT

Amazon's Mechanical Turk ("MTurk") service allows users to post short tasks ("HITs") that other users can receive a small amount of money for completing. Common tasks on the system include labelling a collection of images, combining two sets of images to identify people which appear in both, or extracting sentiment from a corpus of text snippets. Designing a workflow of various kinds of HITs for filtering, aggregating, sorting, and joining data sources together is common, and comes with a set of challenges in optimizing the cost per HIT, the overall time to task completion, and the accuracy of MTurk results. We propose Qurk, a novel query system for managing these workflows, allowing crowd-powered processing of relational databases. We describe a number of query execution and optimization challenges, and discuss some potential solutions.

1. INTRODUCTION

Amazon's Mechanical Turk service (`https://www.mturk.com/mturk/welcome`) ("MTurk") allows users to post short tasks ("HITs") that other users ("turkers") can receive a small amount of money for completing. A HIT creator specifies how much he or she will pay for a completed task. Example HITs involve compiling some information from the web, labeling the subject of an image, or comparing two documents. More complicated tasks, such as ranking a set of ten items or completing a survey are also possible. MTurk is used widely to perform data analysis tasks which are either easier to express to humans than to computers, or for which there aren't yet effective artificial intelligence algorithms.

Task prices vary from a few cents ($.01-$.02/HIT is a common price) to several dollars for completing a survey. MTurk has around 100,000-300,000 HITs posted at any time (`http://mturk-tracker.com/general/`). Novel uses of MTurk include matching earthquake survivor pictures with missing persons in Haiti (`http://app.beextra.org/mission/show/missionid/605/mode/do`), authoring a picture book (`http://bjoern.org/projects/catbook/`), and embedding turkers as editors into a word processing system [2].

From the point of view of a database system, crowd-powered tasks can be seen as operators—similar to user-defined functions—that are invoked as a part of query processing. For example, given a database storing a table of images, a user might want to query for images of flowers, generating a HIT per image to have turkers perform the filter task. Several challenes arise in processing such a workflow. First, each HIT can take minutes to return, requiring an asynchronous query executor. Second, in addition to considering time, a crowdworker-aware optimizer must consider monetary cost and result accuracy. Finally, the selectivity of operators can not be predicted *a priori*, requiring an adaptive approach to query processing.

In this paper, we propose Qurk, a crowdworker-aware database system which addresses these challenges. Qurk can issue HITs that extract, order, filter, and join complex datatypes, such as images and large text blobs. While we describe Qurk here using the language of MTurk (turkers and HITs), and our initial prototype runs on MTurk, we aim for Qurk to be crowd-platform-independent. Future versions of Qurk will compile tasks for different kinds of crowds with different interfaces and incentive systems. Qurk is a new system in active development; we describe our vision for the system, propose a high level sketch of a UDF-like syntax for executing these HITs, and explore a number of questions that arise in such a system, including:

- What is the HIT equivalent of various relational operations? For example, an equijoin HIT might require humans to identify equal items, whereas a HIT-based sort can use human comparators.
- How many HITs should be generated for a given task? For example, when sorting, one can generate a HIT that asks users to score many items, or to simply compare two. How can the system optimize each HIT?
- Given that a large database could result in millions of HITs, how should the system choose which HITs to issue? Given that only a fraction of items in a large database can have HITs generated for them, what is the proper notion of query correctness?
- How much to charge for a HIT? Higher prices can lead to faster answers. Can the system adaptively optimize the price of HITs based on user-specified response time and budget constraints?
- Who is the right crowdworker for a task? MTurk provides paid workers, but an enterprise might prefer expert workers, and an operation with a tight budget might look for non-monetary incentives.

2. MOTIVATING EXAMPLES

Here is a list of tasks that Qurk should be able to run:

CIDR 2011

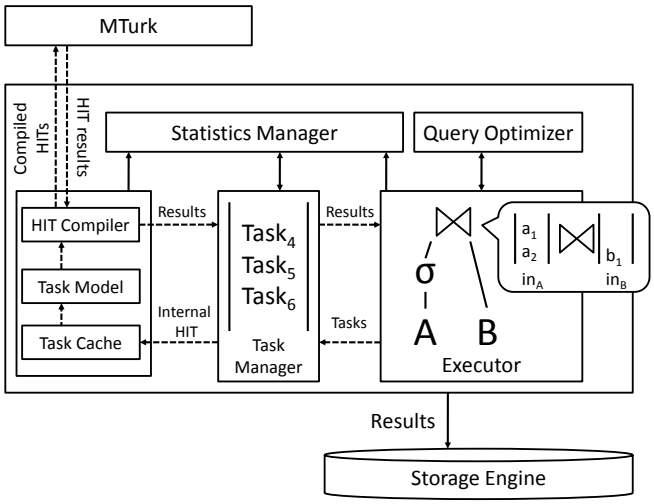


Figure 1: A Qurk system diagram.

- Given a database with a list of company names, find the CEO and contact number for each company (see `http://www.readwriteweb.com/enterprise/2010/01/crowdsourcing-for-business.php`).
- Given a set of photographs of people from a disaster, and pictures submitted by family members of lost individuals, perform a fuzzy join across both sets, using humans to determine equality.
- Given a collection of messages (e.g., from twitter), identify the sentiment of each (e.g., as either "positive" or "negative") [3].
- Given a list of products, rank them by "quality" by searching for reviews on Amazon.

These workflows have relatively small query plans, but in practice, plans see size increases as more filters and sorts appear. Additionally, even small plans benefit from Qurk's adaptive cost optimizations.

3. SYSTEM OVERVIEW

Qurk is architected to handle an atypical database workload. Workloads rarely encounter hundreds of thousands of tuples, and an individual operation, encoded in a HIT, takes several minutes. Components of the system operate asynchronously, and the results of almost all operations are saved to avoid re-running unnecessary steps. We now discuss the details of Qurk, which is depicted in Figure 1.

The **Query Executor** takes as input query plans from the query optimizer and executes the plan, possibly generating a set of tasks for humans to perform. Due to the latency in is processing HITs, the query operators communicate asynchronously through input queues, as in the Volcano system [4]. The example join operator maintains an input queue for each child operator, and creates tasks that are sent to the **Task Manager**.

The Task Manager maintains a global queue of tasks to perform that have been enqueued by all operators, and builds an internal representation of the HIT required to fulfill a task. The manager takes data which the **Statistics Manager** has collected to determine the number of turkers and the cost to charge per HIT, which can differ per operator. Additionally, the manager can collapse several tasks generated by operators into a single HIT. These optimizations collect several tuples from a single operator (e.g., collecting multiple tuples to sort) or from a set of operators (e.g., sending multiple filter operations over the same tuple to a single turker).

A HIT that is generated from the task manager first probes the **Task Cache** and **Task Model** to determine if the the result has been cached (if an identical HIT was executed already) or if a learning model has been primed with enough training data from other HITs to provide a sufficient result without calling out to a crowdworker. We discuss these optimizations further in Section 6.

If the HIT cannot be satisfied by the Task Cache or Task Model, then it is passed along to the **HIT Compiler**, which generates the HTML form that a turker will see when they accept the HIT, as well as other information required by the MTurk API. The compiled HIT is then passed to **MTurk**. Upon completion of a HIT, the Task Model and Task Cache are updated with the newly learned data, and the Task Manager enqueues the resulting data in the next operator of the query plan. Once results are emitted from the topmost operator, they are added to the database, which the user can check on periodically.

4. DATA MODEL AND QUERY LANGUAGE

Qurk's data model is close to the relational model, with a few twists. Two turkers may provide different responses to the same HIT, and results must often be verified for confidence across multiple turkers. In Qurk's data model, the result of multiple HIT answers is a multi-valued attribute. Qurk provides several convenience functions (e.g, `majorityVote`) to convert multi-valued attributes to usable single-valued fields.

We now propose a simple UDF-like approach for integrating SQL with crowdworker-based expressions. We use SQL since it should be familiar to a database audience, but we plan to study a variety of different interfaces in Qurk, some of which will have a more imperative flavor, or which will operate over files instead of database tables. We introduce our proposal through a series of examples.

Filters

In our first example, we look at a query that uses the crowd to find images of flowers in a table called `images`, with an image-valued field called `img`. The user writes a simple SQL query, and uses a UDF called `isFlower`, which is a single-valued function that takes an image as an argument.

```
SELECT * FROM images where isFlower(img)
```

Using the Qurk UDF language, the user defines `isFlower` as the following task:

```
TASK isFlower(Image img) RETURN Bool:
    TaskType: Question
    Text: ''Does this image: <img src='%s'>
          contain a flower?'',URLify(img)
    Response:  Choice(''YES'',''NO'')
```

In our language, UDFs specify the type signature for the `isFlower` function, as well as a set of parameters that define the job that is submitted. In systems like MTurk, a job looks like an HTML form that the turker must fill out. The

parameters above specify the type and structure of the form that the database will generate. The `TaskType` field specifies that this is a question the user has to answer, and the `Response` field specifies that the user will be given a choice (e.g., a radio button) with two possible values (YES, NO), which will be used to produce the return value of the function. The `Text` field shows the question that will be supplied to the turker; a simple substitution language is provided to parameterize the question. Additionally, a library of utility functions like `URLify` (which converts a database blob object into a URL stored on the web) are provided.

Crowd-provided Data

In this example we show how crowdworkers can supply data that is returned in the query answer. Here, the query is to find the CEO's name and phone number for a list of companies. The query would be:

```
SELECT companyName, findCEO(companyName).CEO,
       findCEO(companyName).Phone
FROM companies
```

Observe that the `findCEO` function is used twice, and that it returns a tuple as a result (rather than just a single value). In this case, the `findCEO` function would be memoized after its first application. We allow a given result to be used in several places, including different queries.

The task for the `findCEO` function follows:

```
TASK findCEO(String companyName)
RETURNS (String CEO,String Phone):
    TaskType:  Question
    Text: ``Find the CEO and the CEO's phone
          number for the company %s'', companyName
    Response: Form((``CEO'',String),
                   (``Phone'',String))
```

This task definition is similar to the previous one, except that the response is a tuple with two unconstrained fields.

Table-valued Ranking Functions

Suppose we want the turker to rank a list of products from their Amazon reviews. Our SQL query might be:

```
SELECT productID, productName FROM products
ORDER BY rankProducts(productName)
```

Here, the `rankProducts` function needs take in a list of products, rather than just a single product, so that the user can perform the comparison between products. The task definition is as follows:

```
TASK rankProducts(String[] prodNames) RETURNS String[]:
    TaskType:  Rank
    Text: ``Sort the following list of products
           by their user reviews on Amazon.com''
    List: prodNames
```

The syntax is similar to the previous tasks, except that the function takes an array-valued argument that contains values from multiple rows of the database. The `TaskType` is specified as `Rank`. Ranking tasks don't accept a specific `Response` field, but just provide a list of items for the user to order—in this case, that list is simply the provided array.

The Qurk system may call `rankProducts` multiple times for a given query, depending on the number of items in the table. Ranking tasks are often decomposed into smaller subtasks. To combine results from subtasks, users can be asked to provide a numeric score, or the system can provide an approximate ordering of results by ensuring that subtasks overlap by a few products.

Table-valued Join Operator

In the final example, we show how the crowd can join two tables. Suppose we have a **survivors** table of pictures of earthquake survivors, and a **missing** table with pictures submitted by family members. We want to find missing persons in the survivors table.

```
SELECT survivors.location, survivors.name
FROM survivors, missing
WHERE imgContains(survivors.image, missing.image)
```

The `imgContains` function needs takes two lists of images to join. The task definition is as follows:

```
TASK imgContains(Image[] survivors, Image[] missing)
RETURNS Bool:
    TaskType:  JoinPredicate
    Text: ``Drag a picture of any <b>Survivors</b>
           in the left column to their matching
           picture in the <b>Missing People</b>
           column to the right.''
    Response: DragColumns("Survivors", survivors,
                          "Missing People", missing)
```

Here, `imgContains` is of type `JoinPredicate`, and takes two table-valued arguments. The task is compiled into a HIT of type `DragColumns` which contains two columns labeled `Survivors` and `Missing People`. Turkers drag matching images from the left column to the right one to identify a match. The number of pictures in each column can change to facilitate multiple concurrent comparisons per HIT.

5. QUERY EXECUTION

As mentioned in Section 3, Qurk's components communicate asynchronously. Each query operator runs in its own thread and consumes tuples from its child in the query tree. For each tuple in its queue (or queues, in the case of joins), it generates a task to be completed by a turker. These tasks are enqueued for the task manager, which runs a thread for pricing HITS and collapsing multiple tasks into a single HIT under the instruction of the optimizer. Multiple HITs will be outstanding concurrently.

After a task is returned from a crowdworker, it is used to train a model for potentially replacing turker tasks and is cached for future reuse. A second thread in the manager then enqueues the result of the task into the parent operator's queue. Finally, if the completed task was generated by the root of the query tree, the manager inserts the result into a results table in the database, which the issuer of the Qurk query can return to at a later time to see the results.

6. OPTIMIZATIONS

A Qurk query can be annotated with several parameters that are used to guide its execution. The first, `maxCost`, specifies maximum amount of money the querier is willing to pay. A query might be infeasible if the number of HITs,

each priced at a penny (the cheapest unit of payment), exceed `maxCost`. `minConfidence` specifies the minimum number of turkers who must agree on an answer before it is used. `maxLatency` specifies the maximum amount of time that the user is willing to wait for a HIT to complete.

We plan to explore a cost model based on these parameters in the future. The parameters will help the optimizer identify the cost per HIT, since higher prices decrease the turker wait time. The parameters also affect the number of items batched into a table-valued HIT, such as placing 10 items to a `Rank` task, or 5 items in each column of a `JoinPredicate` task. Finally, the parameters dictate how many times each HIT can be verified by a subsequent turker.

We now discuss several potential optimizations.

Runtime Pricing: If it appears that one type of task takes longer to complete than others, Qurk can offer more money for it in order to get a faster result.

Input Sampling: For large tables that could lead to many HITs, Qurk attempts to sample the input tables to generate Qurk jobs that uniformly cover the input space. This is similar to issues that arise in online query processing [5].

Batch Predicates: When a query contains two filters on the same table, we can combine them into a single HIT, decreasing cost and time. For example, a query with predicates `imgColor(image) == 'Red'` and `imgLabel(image) == 'Car'` could have a single HIT in which turkers are presented with the `imgColor` and `imgLabel` tasks at the same time.

Operator Implementations: To assist with operator pipelining and provide users with results as early as possible, operators should be non-blocking when possible (e.g., symmetric hash joins). Additionally, several potential implementations of each operator are possible. For example, a `Rank` task might ask users to sort a batch of items relative to one-another, or it might ask users to assign an absolute score to each and sort items based on their scores.

Join Heuristics: The space of comparisons required for JoinPredicates can be reduced with a preprocessing step identifying necessary join conditions. For example, if `gender` and `race` must match on pictures of `survivors` and `missing` persons, a user could add a `NecessaryConditions` statement to the `imgContains` task in Section 4 with those two fields. Qurk would generate HITs for each image which instructs turkers to extract `gender` and `race` properties from pictures where possible. `JoinPredicate` HITs could then only compare items that are compatible, reducing the search space.

Task Result Cache: Once a HIT has run, its results might be relevant in the future. For example, if the `products` table has already been ranked in one query, and another query wishes to rank all red `products`, the result of the old HITs can be used for this. Additionally, as explored in TurKit [6], queries might crash midway, or might be iteratively developed. In such scenarios, a cache of completed HITs can improve response time and decrease costs.

Model Training: There are situations where an otherwise acceptable learning algorithm requires training data to be useful. For example, the `isFlower` task in Section 4 could potentially be replaced by an automated classifier with enough training data. If we consider HIT responses as ground truth data, we can incrementally train an algorithm and measure its performance relative to the labeled data set, and then automatically switch over to that algorithm when its performance becomes good enough.

7. RELATED WORK

Little et al. present TurKit [6], which introduces the notion of iterative crowd-powered tasks. TurKit supports a process in which a single task, such as sorting or editing text, might require multiple coordinated HITs, and offers a persistence layer that makes it simple to iteratively develop such tasks without incurring excessive HIT costs.

Several systems have been built on top of MTurk's API to facilitate higher-order tasks using HITs. Crowdflower [1] provides an API to make developing HITs and finding cheating turkers easier, and has abstracted the notion of HITs so that other systems, such as gaming platforms, can also be used to deliver HITs to willing participants.

Parameswaran et al. propose a declarative language for querying crowd workers and outline considerations for an uncertainty model over the results [7]. With Qurk, we instead focus on how to implement crowd-powered operators and design a system for executing queries with these operators. We then describe the optimizations for reducing the cost and time to execute these queries.

8. CONCLUSION

The MTurk ecosystem is growing daily, and a system for building and optimizing data processing workflows across crowdworkers presents many challenges with which the database community is familiar. We proposed Qurk, a system for writing SQL-like queries to manage complex turker workflows, and discussed several challenges and optimization opportunities for the future.

9. REFERENCES

[1] Crowdflower, July 2010. `http://crowdflower.com/`.

[2] M. S. Bernstein et al. Soylent: a word processor with a crowd inside. In *UIST '10: Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 313–322, New York, NY, USA, 2010.

[3] N. A. Diakopoulos and D. A. Shamma. Characterizing debate performance via aggregated twitter sentiment. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 1195–1198, New York, NY, USA, 2010. ACM.

[4] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[5] P. J. Haas et al. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci.*, 52(3):550–569, 1996.

[6] G. Little et al. Turkit: human computation algorithms on mechanical turk. In *UIST '10: Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 57–66, 2010.

[7] A. Parameswaran and N. Polyzotis. Answering queries using databases, humans and algorithms. Technical report, Stanford University.

Enabling Privacy in Provenance-Aware Workflow Systems

Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen
University of Pennsylvania, Philadelphia, USA
{susan, sanjeev, sudeepa, jstoy, val}@seas.upenn.edu

Yi Chen
Arizona State University, Tempe, USA
yi@asu.edu

Tova Milo
Tel Aviv University, Tel Aviv, Israel
milo@post.tau.ac.il

1. INTRODUCTION

A new paradigm for creating and correcting scientific analyses is emerging, that of *provenance-aware* workflow systems. In such systems, repositories of workflow specifications and of provenance graphs that represent their executions will be made available as part of scientific information sharing. This will allow users to search and query both workflow specifications and their provenance graphs: Scientists who wish to perform new analyses may search workflow repositories to find specifications of interest to reuse or modify. They may also search provenance information to understand the meaning of a workflow, or to debug a specification. Finding erroneous or suspect data, a user may then ask provenance queries to determine what downstream data might have been affected, or to understand how the process failed that led to creating the data. With the increased amount of available provenance information, there is a need to efficiently *search* and *query* scientific workflows and their executions.

However, workflow authors or owners may wish to keep some information in the repository confidential. For example, intermediate *data* within an execution may contain sensitive information, such as a social security number, a medical record, or financial information about an individual. Although users with the appropriate access level may be allowed to see such confidential data, making it available to all users, even for scientific purposes, is an unacceptable breach of privacy. Beyond data privacy, a *module* itself may be proprietary, and hiding its description may not be enough: users without the appropriate access level should not be able to *infer* its behavior if they are allowed to see the inputs and outputs of the module. Finally, details of how certain modules in the workflow are connected may be proprietary, and so showing how data is passed between modules may reveal too much of the *structure* of the workflow. **There is thus an inherent tradeoff between the utility of the information provided in response to a search/query and the privacy guarantees that authors/owners desire.**

Scientific workflows are gaining wide-spread use in life sciences applications, a domain in which privacy concerns are particularly acute. We now illustrate three types of privacy using an example from this domain. Consider a personalized disease susceptibility workflow in Fig. 1. Information such as an individual's genetic make-up and family history of disorders, which this workflow takes as input, is highly sensitive and should not be revealed to an unauthorized user, placing stringent requirements on data privacy. Further, a workflow module may compare an individual's genetic make-up to profiles of other patients and controls. The manner in which such historical data is aggregated and the comparison is made, is highly sensitive, pointing to the need for module privacy. Finally, the fact that disease susceptibility predictions are generated by "calibrating" an individual's profile against profiles of others may need to be hidden, requiring that workflow structure be kept private.

As recently noted in [8], "You are better off designing in security and privacy ... from the start, rather than trying to add them later."[1] We apply this principle by proposing that privacy guarantees should be *integrated* in the design of the search and query engines that access provenance-aware workflow repositories. Indeed, the alternative would be to create multiple repositories corresponding to different levels of access, which would lead to inconsistencies, inefficiency, and a lack of flexibility, affecting the desired techniques.

This paper focuses on *privacy-preserving* management of *provenance-aware* workflow systems. We consider the formalization of privacy concerns, as well as query processing in this context. Specifically, we address issues associated with *keyword-based search* as well as with *querying* such repositories for *structural patterns*.

To give some background on provenance-aware workflow systems, we first describe the common model for workflow specifications and their executions (Sec. 2). We then enumerate privacy concerns (Sec. 3), consider their effect on query processing, and discuss the challenges (Sec. 4).

2. MODEL

Workflow *specifications* are typically represented by graphs, with nodes denoting modules and edges indicating dataflow between modules. Workflow specifications may be *hierarchical*, in the sense that a module may be *composite* and itself

[1]While the context for this statement was the use of full body scanning in airports (where the privacy issues are obvious), it is equally valid in provenance systems!

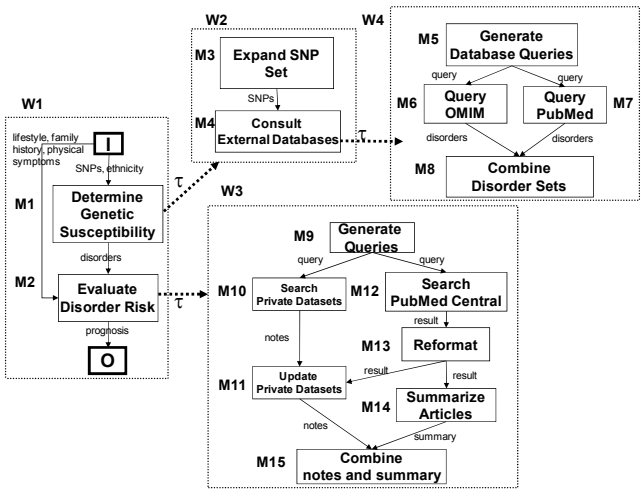


Figure 1: Disease Susceptibility Workflow Specification

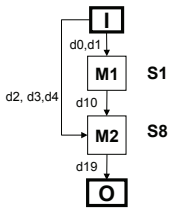


Figure 2: View of Provenance Graph

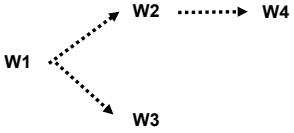


Figure 3: Expansion Hierarchy

contain a workflow. Composite modules are frequently used to simplify workflow design and allow component reuse.

For example, the workflow in Fig. 1 estimates disease susceptibility based on genome-wide SNP array data. The input to the workflow, whose top-most level is given by the dotted box labeled $W1$, is a set of SNPs, ethnicity information, lifestyle, family history, and physical symptoms. The first module in $W1$, M_1, determines a set of disorders the patient is genetically susceptible to based on the input SNPs and ethnicity information. The second module, M_2, refines the set of disorders for which the patient is at risk, based on lifestyle, family history, and physical symptoms.

Fig. 1 also contains τ-labeled edges that give the definitions of composite modules, which we call *expansions*. For example, M_1 is defined by the workflow $W2$, M_2 by the workflow $W3$, and $M4$ by the workflow $W4$. Hence $W2$ and $W4$ are *subworkflows* of $W1$, and $W3$ is a subworkflow of $W2$. The τ expansions (subworkflow relationships) naturally yield an *expansion hierarchy* as shown in Fig. 3.

Prefixes of the expansion hierarchy can be used to define views of a workflow specification.[2] Given a prefix, the *view* that it defines is given by expanding the root workflow so that composite modules in the prefix are replaced by their expansions. For example, consider the expansion hierarchy in Fig. 3 and its prefix consisting of $\{W1, W2\}$. This prefix determines a view of the specification in Fig. 1, which is the

[2]Recall that a *prefix* of a rooted tree T is a tree obtained from T by deleting some of its subtrees (i.e., some nodes and all their descendants).

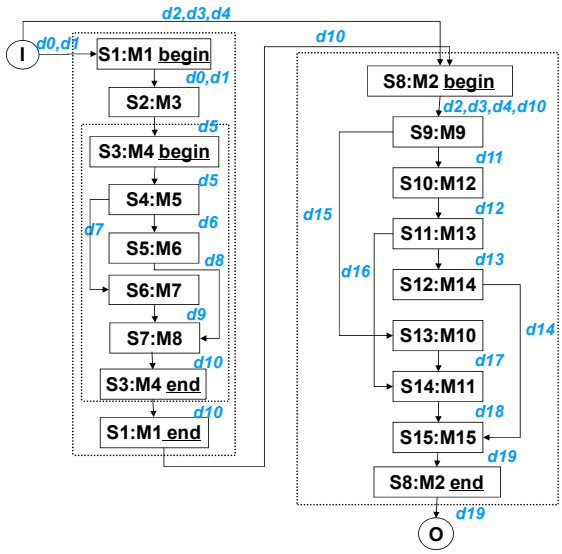


Figure 4: Disease Susceptibility Workflow Execution

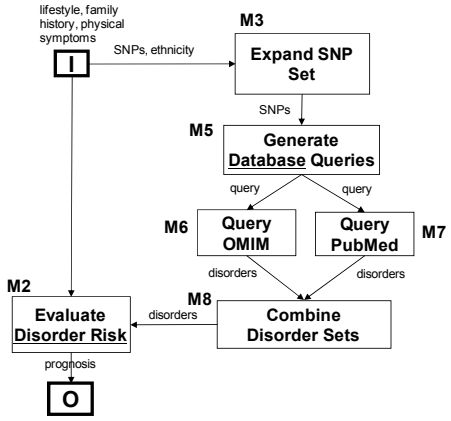


Figure 5: Result of Query "Database, Disorder Risks"

simple workflow obtained from $W1$ by replacing $M1$ with $W2$. Another view is the full expansion, which yields a workflow with module names $I, O, M3$,and $M5 - M15$ and whose edges include one from $M3$ to $M5$ and another from $M8$ to $M9$. We will shortly discuss the benefit of views.

A workflow specification describes the possible run-time *executions*. Executions are modeled similarly to simple workflow graphs, but additionally associate a unique process id with a module execution, and data with edges. When execution reaches a composite module, it continues in the corresponding subworkflow and eventually returns (like a procedure call). For example, an execution of the workflow specification in Fig. 1 is shown in Fig. 4. In this example, for clarity we show the process id appended to the name of the module being executed, e.g. S1:M1. Following common practice [1], each composite module execution is represented by two nodes, the first standing for its activation and the second for its completion, e.g. S1:M1-begin and S1:M1-end.

In an execution, data flows over the edges. We assume that each data item is the output of exactly one module execution and has a unique id. We can therefore annotate each edge $M \rightarrow N$ in the execution with the set of data

items that flow as the output of M to the input of N. For example, in Fig. 4 the set $\{d0, d1\}$ flows from I to S1:M1.

The *provenance* of a data item d in an execution E is therefore the subgraph induced by the set of paths from the start node to the end node of E that produced d as output. In the sequel, we blur the distinction between the provenance of data items and the executions that produce them.

As introduced in [2], we can use views to simplify what is seen of an execution. Using the view defined by prefix $\{W1\}$, the execution of Fig. 4 would be simplified to that in Fig. 2. Thus views can be used to define access control to address privacy concerns. Specifically, we can define a user's access privilege as the finest grained view that s/he can access, called an *access view*.

3. PRIVACY

Privacy concerns are tied to the workflow components: data, modules, and the structure of a workflow. To illustrate them, consider again the sample workflow in Fig. 1. *Data privacy* requires that the output of M1, i.e., the genetic disorders the patient is susceptible to, should not be revealed with high probability, in any execution, to users without the required access privilege. Such data masking is a fairly standard requirement in privacy-aware database systems. *Module privacy* is more particular: It requires that the functionality of a private module – that is, the mapping it defines between inputs and outputs – is not revealed to users without the required access privilege. Returning to our example, assuming that M1 implements a function f_1, module privacy with respect to M1 requires that no adversarial user should be able to guess the output f_1(`SNP`, `ethnicity`) with high probability for any `SNP` and `ethnicity` input. From a patient's perspective, this is important because they do not want someone who may happen to have access to their `SNP` and `ethnicity` information to be able to determine what `disorders` they are susceptible to. From the module owner's perspective, they do not want the module to be simulated by competitors who capture all input-output relationships. Finally, *structural privacy* refers to hiding structure of the information flow in the given execution. In this example it might mean that users without the required access privilege should not know whether or not `lifestyle` was used to calculate the `disorders` output by M1.

Broadly speaking, the fundamental privacy question to be addressed is: **How do we provide provable guarantees on privacy of components in a workflow while maximizing utility with respect to provenance queries?**

In doing so, we must understand 1) how to measure privacy; 2) what information can be hidden; 3) how to measure utility; and 4) how to efficiently find solutions that *simultaneously* provide provably good guarantees on privacy and utility. It is worth highlighting an important characteristic, namely, that all privacy guarantees are required to hold over repeated executions of a workflow with varied inputs.

We discuss module and structural privacy in more depth before turning to the impact of privacy on search and query mechanisms.

Module Privacy. It is easy to see that, if information about all intermediate data is repeatedly given for multiple executions of a workflow on different initial inputs, then partial or complete functionality of modules may be revealed. The approach that we take in [4] is to *hide a carefully chosen subset of intermediate data*, thereby limiting the amount of provenance data shown to the user and guaranteeing some desired level of privacy. Ignoring for now structural privacy, one may assume that users can see the connections (edges) between modules in the workflow; only the *values* of selected intermediate data are hidden, in *all* executions of the workflow. Since there may be several different subsets of intermediate data whose hiding yields the desired level of privacy, and certain data may have higher utility to users than other data (i.e., data may be weighted), this becomes an interesting optimization problem.

Structural Privacy. The goal of structural privacy is to keep private the information that some module M contributes to the generation of a data item d, output by another module M'. For instance, in the execution of the workflow $W3$, we may wish to hide the fact that the reformatted data from PubMed Central (module $M13$) contributes to updating the private DB, and hence to the output of module $M11$. One possible approach is to delete edges and vertices so as to eliminate all paths from M to M', e.g., in this example to delete the edge $M13 \rightarrow M11$. However, by doing so, we may hide additional provenance information that does not need be hidden (e.g., the existence of a path from $M12$ to $M11$). Another approach is to use *clustering*, where certain modules are hidden in a composite module P so that the reachability of any pair (u, v) in P is no longer externally visible. For example, we could cluster $M11$ and $M13$ into a single composite module. However, we may now infer incorrect provenance information, e.g., that there is a path from $M10$ to $M14$. This is called an *unsound view* in [3, 9]. Once again, one faces a challenging optimization problem: guaranteeing an adequate level of privacy while preserving soundness and minimizing unnecessary loss of information.

4. PRIVACY-PRESERVING QUERY EVALUATION

Query languages for workflow specifications/executions support two main types of queries: *structural queries* that allow users to select sub-workflows based on structural properties (e.g., "find executions where `Expand SNP Set` was executed before `Query OMIM` and return the provenance information for the latter") and *keyword queries* that retrieve sub-workflows that match the input keywords (e.g., "find workflows that include 'disorder risk' and 'database'", result shown in Fig. 5). In both cases the query answer is given as a minimal view of the flow that satisfies the query criteria and includes the keywords (see [1, 7] for formal definitions). Much research was recently devoted to developing efficient query evaluation techniques in this context. Unfortunately, none of this works addresses the privacy issues mentioned in the previous section. We consider below the main challenges in enabling such privacy-preserving query evaluation.

Privacy-controlled Semantics for Queries. Before one can consider efficient query evaluation, there is a need to formally define the semantics of queries in this context. What is the correct answer to a given query, assuming privacy and access control settings that are guided by the hierarchical structure of workflow specifications? Notably, the three different kinds of privacy we consider may have different impacts on what information should be available to a given user, and therefore on the semantics of queries and on the definition of search results. For example, consider struc-

tural privacy, which may be achieved by clustering nodes to form a composite module. Such an approach may introduce extraneous paths, causing misleading provenance information [9]. A challenging question to investigate is then the following: In the presence of unsound views, how can we define search results that maximize utility (defined to be some function of both the number of correct node connectivity relationships captured and the number of modules disclosed in a result), while guaranteeing privacy?

Efficient Search with Privacy Guarantees. There is clearly a distinction between defining the formal semantics of relevance and computing the answers efficiently. It may be infeasible to create variants of the workflow repository, one for each privilege/privacy setting, due to high space overhead. Instead, the information must be hidden on-the-fly, which usually leads to processing overhead. A challenge is then to develop algorithms for addressing these computational problems.

First, standard, non-privacy preserving workflow management systems use various indexing structures or materialized views to speed up query processing. With data privacy, we must manage an index with "different user views", as users often have different privileges on data accesses. A promising direction is to consider representing the specification and execution graphs using advanced data structures that classify and group their elements based on privacy settings. Another promising direction is to consider user groups when utilizing cached information during query processing.

Second, to achieve privacy, one needs to generate query results with respect to user access privileges (view). One approach would be to first construct a full answer, oblivious to the privacy requirement. If the result reveals sensitive information, we may gradually "zoom-out" the view by hiding details of composite modules and sensitive data, until privacy is achieved. However, this can be expensive as each zoom-out may involve a disk access. Techniques must be developed to efficiently construct user-specific answers.

Impact of Ranking on Privacy Preservation. Sometimes a user query is ambiguous and the results can have varying degrees of relevance. Ranking is therefore an important function, especially for a keyword-based search engine. One typical metric in ranking is to consider term frequency (TF) and inverse document frequency (IDF). A highly ranked result is likely to have more occurrences of an input keyword than a lowly ranked result. Thus, a user might be able to infer the range of value occurrences in a result even though s/he is unable to see the values due to privacy preservation. Such inference may cause information leakage, and affect module and value privacy. A challenge is to design sophisticated ranking schemes that not only rank results in the order of relevance but are also privacy-aware.

5. RELATED WORK AND CONCLUSIONS

Extensive work has been done on privacy in various settings, e.g., *data mining*, *social networks*, *auditing queries*, and *statistical databases*. The problem of defining a consistent set of *access controls* to preserve privacy in a workflow has also been considered, as well as the problem of ensuring the *lawful use* of data according to some specified privacy policies. However, a formal study of privacy issues specific to workflows, with provable privacy guarantees, has not yet been done. It will be particularly interesting to see if ideas from *differential privacy* [5, 6] can be used in this setting. Although it is the strongest notion of privacy known to date, it is also known that *no* deterministic algorithm can guarantee differential privacy. This may limit the applicability of differential privacy in our setting — provenance in scientific workflows is used to ensure reproducibility of experiments, and adding random noise to provenance information may render it useless.

Keyword search has been extensively studied, e.g., for *graph-structured* and *tree-structured* (e.g., XML) data. There has also been work on *graph query languages*. Nonetheless, prior work does not adequately address the requirements of privacy-aware workflow management systems, for two reasons. First, workflow specifications involve both dataflow and expansion (τ) edges, and the difference between them cannot be ignored [1, 7]. Second, prior work does not consider search and query processing in the face of privacy requirements.

In summary, there are significant novel research challenges that must be addressed in developing the next generation of privacy-enabled provenance-aware workflow systems. These range from formalizing privacy requirements and notions of utility, and developing algorithms for associated optimization problems, to designing efficient systems that integrate privacy with query processing mechanisms.

6. ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation grants IIS-0803524, IIS-0629846, and IIS-0915438, by NSF CAREER award IIS-0845647, and by grant 0937060 to the Computing Research Association for the CIFellows Project. This work was also supported in part by an IBM faculty award. This work was also supported in part by the Israel Science Foundation, and by the US-Israel Binational Science Foundation.

7. REFERENCES

[1] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with BP-QL. *Information Systems*, 33(6):477–507, 2008.

[2] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, pages 1072–1081, 2008.

[3] O. Biton, S. B. Davidson, S. Khanna, and S. Roy. Optimizing user views for workflows. In *ICDT*, pages 310–323, 2009.

[4] S. B. Davidson, S. Khanna, D. Panigrahi, and S. Roy. Preserving module privacy in workflow provenance. *Manuscript available at http://arxiv.org/abs/1005.5543.*

[5] C. Dwork. Differential privacy: A survey of results. In *TAMC*, pages 1–19, 2008.

[6] C. Dwork. The differential privacy frontier (extended abstract). In *TCC*, pages 496–502, 2009.

[7] Z. Liu, Q. Shao, and Y. Chen. Searching workflows with hierarchical views. *PVLDB*, 3(1), 2010.

[8] S. S. Shapiro. Privacy by design: moving from art to practice. *Commun. ACM*, 53(6):27–29, 2010.

[9] P. Sun, Z. Liu, S. B. Davidson, and Y. Chen. Detecting and resolving unsound workflow views for correct provenance analysis. In *SIGMOD*, pages 549–562. ACM, 2009.

The Schema-Independent Database UI

A Proposed Holy Grail and Some Suggestions

Eirik Bakke
MIT CSAIL
ebakke@mit.edu

Edward Benson
MIT CSAIL
eob@mit.edu

ABSTRACT

If you have ever encountered a piece of highly domain-specific business software, you may have noticed that it was largely a graphical front-end to some relational database. You may also, in fact, have avoided using the system at all—studies show that information workers prefer to dump their data into spreadsheets, a general and more familiar tool which, unfortunately, is poorly suited for many standard database tasks. It is time that we stop streamlining the process of creating a new application for every schema, and that we instead develop the visual query languages that will let end-users access the full power of relational database management systems from a simple and unified interface. Once information workers can create, manage, and query real databases with the same ease as they routinely manipulate spreadsheets today, they will never return to their schema-dependent, consultant-made, and oddly-colored Microsoft Access applications.

1. A WAR STORY

In his younger days (well, before undergrad), one of the authors worked as an administrative assistant for one of Norway's 400 public schools of music and arts. To keep track of students, teachers, lessons, and rental instruments, the school had licensed a special-purpose database application, originally developed in FileMaker Pro[1] and later rewritten in 4th Dimension[2]. See Figure 1. The software was made and sold by a six-person consulting firm started by a former band director and FileMaker whiz on the other side of the country, and was constantly under development. A couple of times per year, the consultants would fly out to our individual schools to train us in the use of new features as well as collect feedback for further development. A common kind of request would be adding another field to an entity type in the database; for instance, we might ask if we could "get a checkbox in the 'Student' dialog to indicate whether their parents had signed the audio recording release form yet." The consultants were generally very helpful, and sure enough, three months later there would be a patch released with the new field in the database. One day, however, they balked. "Sorry. There is no more room in the dialog box."

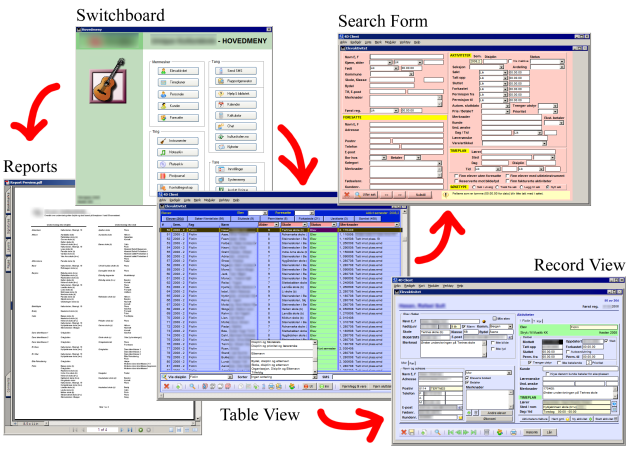


Figure 1: The stereotypical user interface of a FileMaker-style database application. Screenshots from an administration system for public Norwegian music schools.

2. TAILORED DATABASE APPLICATIONS

The story above, which could be retold for thousands of organizations worldwide, illustrates a number of points about database applications "in the wild." First, there is an infinite number of database schemas which may seem perfectly obscure to the world at large, but which may be of great value to a small number of people or organizations (e.g. schemas that deal with the intricacies of running a public Norwegian music school). Second, it takes a lot of effort, and a lot of attention to detail, to implement database applications for production use in such environments. In fact, it may take a small start-up—and we know of companies, such as Visma[3], that make a living out of acquiring said small start-ups after the they begin to reach market saturation. Third, there inevitably ends up being a gap between users and developers, and the schema of the database is hard to change. Moreover, users are no longer fully in control of their data; on one occasion, continuing the story above, we paid the consultants NOK 10,000 (about $2,000) to migrate

[1] http://www.filemaker.com
[2] http://www.4d.com
[3] http://www.visma.com

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

a "record first created at [some date]" field from the old FileMaker database to the newer 4th Dimension one on "only" a week's notice. Fourth, tailor-made database applications require their users to undergo training and learning periods which may be expensive in terms of both the consultants' and the end-users' time. Not to mention customer support: software that is under constant development is unlikely to be free of critical bugs, technical or usability-related. This is true even, and maybe especially so, for large organizations; while a larger number of potential target users may well decrease the marginal cost of custom software development, total training and support costs increase all the more.

A few other things are worth mentioning. Tailor-made database applications, like the one in the story, do very little other than provide basic *CRUD* (Create, Read, Update, Delete) facilities on their underlying relational databases. While the applications often include certain special features hard-coded for the schema in question (for instance, our system could send text messages to students based on their "cell phone number" field), these are typically not the main reason for using the application. Rather, people use database applications because they need to manage many different entities (students, teachers, lessons, instruments... or parts, suppliers, and plants for that matter) and the relationships between them. In other words, these are exactly the kind of tasks that relational databases were made for handling well.

Last, tailor-made applications seldom reach anywhere near the same level of maturity as more general-purpose ones. Features that are taken for granted in other applications may never get implemented in a tailor-made application, simply because the development time would not be justified for the size of the user base. In the music school system example, the developers found the table widget available in 4th Dimension inadequate for their needs, and decided to roll their own (presumably due to the limitations inherent in two-dimensional table views, see our later discussion). It was only after two years that they released a patch to allow the mouse wheel to be used to scroll the table up and down.

3. SPREADSHEETS

With all the problems associated with tailor-made database applications, is there an alternative? Currently, only "sort of."

If given a choice, information workers would rather dump their data into a spreadsheet than use some odd database application, even at substantial pain. Spreadsheet users "shun enterprise solutions" [7] and "do not appear inclined to use other software packages for their tasks, even if these packages might be more suitable" [3]. One survey shows that "sorting and database facilities" are the most commonly used spreadsheet features, with 70% of business professionals using them on a frequent or occasional basis [6]. In contrast, less than half use "tabulation and summary measures such as averages/totals"—one of the original design goals of the original VisiCalc spreadsheet. In fact, one of the most frequent uses of spreadsheets may be as a pseudo-database.

Spreadsheets have some great properties. Their interfaces are extremely mature and afford a large range of streamlined facilities for working with any data that can be arranged in a grid of cells, including multiple selection, copy/paste, find/replace, undo/redo, inserting and deleting, extending data values, sorting and filtering on arbitrary fields, navigating and selecting cells with the keyboard, and so on. They have hundreds of millions of users worldwide that are already trained in their use, and a relatively large portion of these are power users. And, of course, there are no development costs.

Unfortunately, spreadsheets lack features essential to database applications, including joins, views, forms, report generation, multiple users, and so on. In effect, spreadsheets are great for single-table databases shared between only a few users, but become very painful to use as tasks scale out in various dimensions.

4. THE HOLY GRAIL

Spreadsheet software is the quintessential example of a *general-purpose data manipulation tool*, and the fact that people use it for database tasks despite great pains should be an indicator that such tools are the way to go. If we could make a general-purpose data manipulation tool that covers even, say, 80% of the functionality afforded by typical tailor-made database applications, information workers would never look back.

We should give credit to others who have proposed similar grails before. Quoting Yannis Ioannidis:

"Visual database query languages that can express the full spectrum of complex queries desired and data visualization mechanisms that can capture the essence of large and complex data sets in ways that match users' intuition are somewhat of a Holy Grail in database user interfaces. Unfortunately, almost nobody seems to be looking for it!" [4]

Visual query languages are indeed central to the idea of of making a general-purpose data manipulation tool. A tool that is intended to replace the majority of tailor-made database applications must certainly give the user the power to express SQL-like queries from a graphical interface. In fact, we need more than relational completeness, since many of the views desired in a database front-end are hierarchical rather than tabular; we discuss this below. Now quoting the 2005 The Lowell Database Research Self-Assessment:

"It is a long-standing lament that the database community does too little about user interfaces. (...) A small number of slick visualization systems oriented toward information presentation were proposed during the 1980s, notably QBE and VisiCalc. There have not been comparable advances in the last 15 years, and there is a substantial need for better ideas in this area." [1]

It is interesting to note that one of the two notable visual query languages cited is VisiCalc: the spreadsheet itself. Keep in mind, spreadsheets are indeed one kind of visual query language. We need better ones, so that they can replace our tailor-made database applications.

5. THE STATE OF THE ART

Since the stereotypical user interface design of a tailor-made database application follows rather mechanically from the underlying database schema (see again Figure 1), application builders like FileMaker et al. provide plenty of shortcuts and wizards to create them, as do development frameworks such as Ruby on Rails[4]. Systems like AppForge [9] and App2You [5] take these ideas further than their predecessors, abstracting away much of the low-level form design

[4] `http://guides.rubyonrails.org/getting_started.html#getting-up-and-running-quickly-with-scaffolding`

and data binding work required earlier, making the application development process more WYSIWYG, and hiding the technical details of relational schema design from the user. Still, these systems are "application builders"—rather than aiming to be general data management applications that can be used with any schema, they aim to make it easier for developers to churn out special-purpose ones at a faster pace. The user interfaces produced are just as schema-dependent as before, they lack good features for general-purpose data management, and users have to learn to use them from scratch.

Despite being in the application builder category, we consider AppForge [9] to have made a major step in the right direction, by creating a visual query language that allows the user to retrieve joined hierarchical views of the data in the database. We believe such a language must be a core part of any general-purpose data manipulation tool that intends to replace tailor-made database applications. Query-by-Example was already mentioned as a visual query language [10, 1]; however, it is only able to retrieve flat tables similar to those returned by SQL queries, and is as such not expressive enough for our purposes. A major and relatively recent visual query language is that of Polaris/Tableau [8]; this system provides a very expressive user interface for defining visualizations and aggregations on tabular or multidimensional data, based on the pivot table concept found originally in Lotus Improv (1993). The output, however, remains in flat table form, possibly rendered on screen using a selection of visualizations. Pivot table systems provide cross-tabulated, as opposed to hierarchical, views of the user's data.

6. SOME SUGGESTIONS

What would a universal tool for relational data look like? It would likely be a blend of interface norms from the spreadsheet world along with more powerful data management capabilities from the database world. In particular, we imagine:

- The tool is not a builder for the interface. The tool *is* the data interface. Users should be able to construct the particular views they need as they go, at a small enough cost that these views can be treated as disposable objects.
- The tool should allow users to work with data as a set of views. The underlying relational structure should be available for power-users, but hidden by default.
- The tool can read and write both data and schemas. A flexible, universal interface for existing relational schemas would be an achievement in its own right, but we believe such tools should be able to create and modify schemas as well. Users, however, should not need to be aware of these operations unless they want to be.
- The tool should present data to the user as complex structures where appropriate, even if the data is not displayed as such relationally. One-to-many or many-to-many relationships may, for instance, appear as a bulleted list from the perspective of a particular object of focus.
- The tool should contain a visual query language. That is, it allows the user to construct and compose view queries over the data using a graphical user interface.
- As a necessity, the tool must have features common to spreadsheets (multiple selection, search/replace, etc) and existing database applications (form-style browsing, reports, etc).
- The tool errs on the side of off-the-shelf usability, requiring technical input from users only when necessary to resolve a potential ambiguity. Calendar applications provide a good example of this: when an event in a repeated series is modified, the application provides the user with several options: modify one item, modify the entire series, or modify all future events. Relationships between entities create many such ambiguous situations that should be handled as conversations between the computer and the user rather than by requiring the user to specify *a priori* how she intends to modify information in the future.

We propose that a universal tool may be constructed by relying on hierarchical views as the interface metaphor. The key insight is that hierarchical views seem to describe the set of structures that humans are interested in looking at. Look at almost any user interface in a database-backed application, and you will observe a hierarchy:

- Patient records in a medical database, with doctor visits as an embedded table of each patient
- Course records in a school database, with required readings and teaching staff as embedded tables
- Emails in an inbox, decorated with tags, subject, and sender name
- Books in a bookstore, displayed by category

Given this observation, the role of the small-business interface database consultant appears to be writing code to translate a relational programming interface into a series of hierarchical views. If what we want are hierarchical views, then *build databases that allow us to pretend the information is hierarchical in the first place.*

Building database tools which embrace the commonality of hierarchical information browsing would enable the creation of databases that interact with casual users in the same way as they think about their data. At any given time, most users want to look at an entity, or a list of entities, along with properties and possibly related other entities. SQL can always be used in the exceptional cases.

7. RELATED WORKSHEETS

One of our own systems, Related Worksheets [2], attempts to explore the spreadsheets-as-a-database concept by extending the spreadsheet paradigm to let the user establish relationships between rows in related worksheets as well as view and navigate the hierarchical cell structure that arises as a result. A user study on 36 regular Excel users showed that first-time users of our system were able to solve lookup-style query tasks with the same or better accuracy than subjects in a control group using Excel, in one case 40% faster on average ($p<0.05$). See Figure 2.

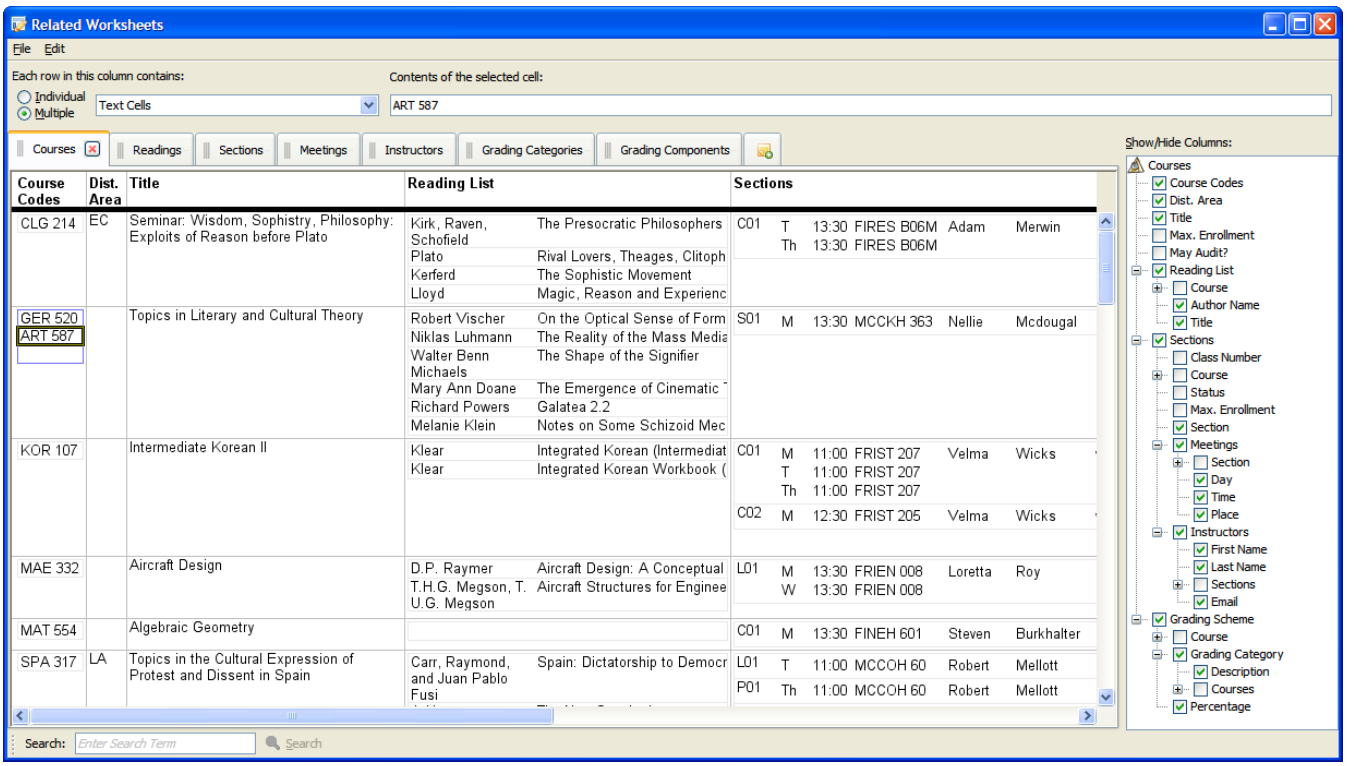


Figure 2: Screenshot of our Related Worksheets application, showing a hierarchical view of entities related through one-to-many and many-to-many relationships.

8. CONCLUSION

Since the early years of database management systems, we have kept building a new database application for every schema that came along. These resulting *tailor-made* database applications are expensive to develop, hard to maintain, hard to use, and hardly much more than graphical front-ends to their underlying databases. By looking to the success of spreadsheets as a general-purpose data management tool, and by developing new visual query languages to let end-users access the full power of relational database management systems from a simple and unified interface, we can eliminate the pains of using either spreadsheets or tailor-made applications for database tasks, and get the best of both worlds.

9. REFERENCES

[1] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, J. Gray, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, M. Kersten, M. Pazzani, M. Lesk, D. Maier, J. Naughton, H. Schek, T. Sellis, A. Silberschatz, M. Stonebraker, R. Snodgrass, J. Ullman, G. Weikum, J. Widom, and S. Zdonik. The Lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[2] E. Bakke, D. R. Karger, and R. C. Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *CHI*, 2011 (to appear).

[3] Y. E. Chan and V. C. Storey. The use of spreadsheets in organizations: determinants and consequences. *Information & Management*, 31(3):119–134, 1996.

[4] Y. E. Ioannidis. Visual user interfaces for database systems. *ACM Comput. Surv.*, page 137, 1996.

[5] K. Kowalzcykowski, A. Deutsch, K. W. Ong, Y. Papakonstantinou, K. K. Zhao, and M. Petropoulos. Do-It-Yourself database-driven web applications. In *CIDR*, 2009.

[6] J. D. Pemberton and A. J. Robson. Spreadsheets in business. *Industrial Management & Data Systems*, 200(8):379–388, 2000.

[7] N. Raden. Shedding light on shadow IT: Is Excel running your business? Technical report, Hired Brains, Inc., January 2005.

[8] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[9] F. Yang, N. Gupta, C. Botev, E. F. Churchill, G. Levchenko, and J. Shanmugasundaram. WYSIWYG development of data driven web applications. *Proc. VLDB Endow.*, 1(1):163–175, 2008.

[10] M. M. Zloof. Query-by-Example: A data base language. *IBM Syst. J.*, 16(4):324–343, 1977.

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

{jasonbaker,chrisbond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

ABSTRACT

Megastore is a storage system developed to meet the requirements of today's interactive online services. Megastore blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS in a novel way, and provides both strong consistency guarantees and high availability. We provide fully serializable ACID semantics within fine-grained partitions of data. This partitioning allows us to synchronously replicate each write across a wide area network with reasonable latency and support seamless failover between datacenters. This paper describes Megastore's semantics and replication algorithm. It also describes our experience supporting a wide range of Google production services built with Megastore.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases*

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Large databases, Distributed transactions, Bigtable, Paxos

1. INTRODUCTION

Interactive online services are forcing the storage community to meet new demands as desktop applications migrate to the cloud. Services like email, collaborative documents, and social networking have been growing exponentially and are testing the limits of existing infrastructure. Meeting these services' storage demands is challenging due to a number of conflicting requirements.

First, the Internet brings a huge audience of potential users, so the applications must be *highly scalable*. A service can be built rapidly using MySQL [10] as its datastore, but scaling the service to millions of users requires a complete redesign of its storage infrastructure. Second, services must compete for users. This requires *rapid development* of features and fast time-to-market. Third, the service must be responsive; hence, the storage system must have *low latency*. Fourth, the service should provide the user with a *consistent view of the data*—the result of an update should be visible immediately and durably. Seeing edits to a cloud-hosted spreadsheet vanish, however briefly, is a poor user experience. Finally, users have come to expect Internet services to be up 24/7, so the service must be *highly available*. The service must be resilient to many kinds of faults ranging from the failure of individual disks, machines, or routers all the way up to large-scale outages affecting entire datacenters.

These requirements are in conflict. Relational databases provide a rich set of features for easily building applications, but they are difficult to scale to hundreds of millions of users. NoSQL datastores such as Google's Bigtable [15], Apache Hadoop's HBase [1], or Facebook's Cassandra [6] are highly scalable, but their limited API and loose consistency models complicate application development. Replicating data across distant datacenters while providing low latency is challenging, as is guaranteeing a consistent view of replicated data, especially during faults.

Megastore is a storage system developed to meet the storage requirements of today's interactive online services. It is novel in that it blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS. It uses synchronous replication to achieve high availability and a consistent view of the data. In brief, it provides fully serializable ACID semantics over distant replicas with low enough latencies to support interactive applications.

We accomplish this by taking a middle ground in the RDBMS vs. NoSQL design space: we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We provide traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support. We contend that the data for most Internet services can be suitably partitioned (e.g., by user) to make this approach viable, and that a small, but not spartan, set of features can substantially ease the burden of developing cloud applications.

Contrary to conventional wisdom [24, 28], we were able to use Paxos [27] to build a highly available system that pro-

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

vides reasonable latencies for interactive applications while synchronously replicating writes across geographically distributed datacenters. While many systems use Paxos solely for locking, master election, or replication of metadata and configurations, we believe that Megastore is the largest system deployed that uses Paxos to replicate primary user data across datacenters on every write.

Megastore has been widely deployed within Google for several years [20]. It handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of primary data across many global datacenters.

The key contributions of this paper are:

1. the design of a data model and storage system that allows rapid development of interactive applications where high availability and scalability are built-in from the start;
2. an implementation of the Paxos replication and consensus algorithm optimized for low-latency operation across geographically distributed datacenters to provide high availability for the system;
3. a report on our experience with a large-scale deployment of Megastore at Google.

The paper is organized as follows. Section 2 describes how Megastore provides availability and scalability using partitioning and also justifies the sufficiency of our design for many interactive Internet applications. Section 3 provides an overview of Megastore's data model and features. Section 4 explains the replication algorithms in detail and gives some measurements on how they perform in practice. Section 5 summarizes our experience developing the system. We review related work in Section 6. Section 7 concludes.

2. TOWARD AVAILABILITY AND SCALE

In contrast to our need for a storage platform that is global, reliable, and arbitrarily large in scale, our hardware building blocks are geographically confined, failure-prone, and suffer limited capacity. We must bind these components into a unified ensemble offering greater throughput and reliability.

To do so, we have taken a two-pronged approach:

- for availability, we implemented a synchronous, fault-tolerant log replicator optimized for long distance-links;
- for scale, we partitioned data into a vast space of small databases, each with its own replicated log stored in a per-replica NoSQL datastore.

2.1 Replication

Replicating data across hosts within a single datacenter improves availability by overcoming host-specific failures, but with diminishing returns. We still must confront the networks that connect them to the outside world and the infrastructure that powers, cools, and houses them. Economically constructed sites risk some level of facility-wide outages [25] and are vulnerable to regional disasters. For cloud storage to meet availability demands, service providers must replicate data over a wide geographic area.

2.1.1 Strategies

We evaluated common strategies for wide-area replication:

Asynchronous Master/Slave A master node replicates write-ahead log entries to at least one slave. Log appends are acknowledged at the master in parallel with transmission to slaves. The master can support fast ACID transactions but risks downtime or data loss during failover to a slave. A consensus protocol is required to mediate mastership.

Synchronous Master/Slave A master waits for changes to be mirrored to slaves before acknowledging them, allowing failover without data loss. Master and slave failures need timely detection by an external system.

Optimistic Replication Any member of a homogeneous replica group can accept mutations [23], which are asynchronously propagated through the group. Availability and latency are excellent. However, the global mutation ordering is not known at commit time, so transactions are impossible.

We avoided strategies which could lose data on failures, which are common in large-scale systems. We also discarded strategies that do not permit ACID transactions. Despite the operational advantages of eventually consistent systems, it is currently too difficult to give up the read-modify-write idiom in rapid application development.

We also discarded options with a heavyweight master. Failover requires a series of high-latency stages often causing a user-visible outage, and there is still a huge amount of complexity. Why build a fault-tolerant system to arbitrate mastership and failover workflows if we could avoid distinguished masters altogether?

2.1.2 Enter Paxos

We decided to use Paxos, a proven, optimal, fault-tolerant consensus algorithm with no requirement for a distinguished master [14, 27]. We replicate a write-ahead log over a group of symmetric peers. Any node can initiate reads and writes. Each log append blocks on acknowledgments from a majority of replicas, and replicas in the minority catch up as they are able—the algorithm's inherent fault tolerance eliminates the need for a distinguished "failed" state. A novel extension to Paxos, detailed in Section 4.4.1, allows local reads at any up-to-date replica. Another extension permits single-roundtrip writes.

Even with fault tolerance from Paxos, there are limitations to using a single log. With replicas spread over a wide area, communication latencies limit overall throughput. Moreover, progress is impeded when no replica is current or a majority fail to acknowledge writes. In a traditional SQL database hosting thousands or millions of users, using a synchronously replicated log would risk interruptions of widespread impact [11]. So to improve availability and throughput we use multiple replicated logs, each governing its own partition of the data set.

2.2 Partitioning and Locality

To scale our replication scheme and maximize performance of the underlying datastore, we give applications fine-grained control over their data's partitioning and locality.

2.2.1 Entity Groups

To scale throughput and localize outages, we partition our data into a collection of *entity groups* [24], each independently and synchronously replicated over a wide area. The underlying data is stored in a scalable NoSQL datastore in each datacenter (see Figure 1).

Entities within an entity group are mutated with single-phase ACID transactions (for which the commit record is

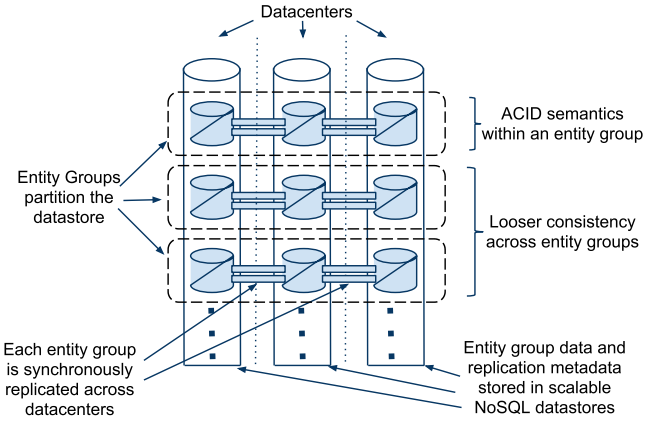


Figure 1: Scalable Replication

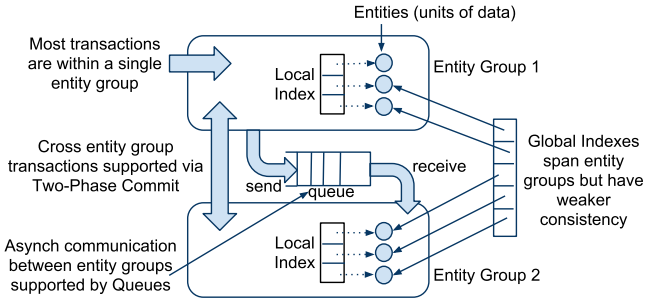


Figure 2: Operations Across Entity Groups

replicated via Paxos). Operations across entity groups could rely on expensive two-phase commits, but typically leverage Megastore's efficient asynchronous messaging. A transaction in a sending entity group places one or more messages in a queue; transactions in receiving entity groups atomically consume those messages and apply ensuing mutations.

Note that we use asynchronous messaging between logically distant entity groups, not physically distant replicas. All network traffic between datacenters is from replicated operations, which are synchronous and consistent.

Indexes local to an entity group obey ACID semantics; those across entity groups have looser consistency. See Figure 2 for the various operations on and between entity groups.

2.2.2 *Selecting Entity Group Boundaries*

The entity group defines the *a priori* grouping of data for fast operations. Boundaries that are too fine-grained force excessive cross-group operations, but placing too much unrelated data in a single group serializes unrelated writes, which degrades throughput.

The following examples show ways applications can work within these constraints:

Email Each email account forms a natural entity group. Operations within an account are transactional and consistent: a user who sends or labels a message is guaranteed to observe the change despite possible failover to another replica. External mail routers handle communication between accounts.

Blogs A blogging application would be modeled with multiple classes of entity groups. Each user has a profile, which is naturally its own entity group. However, blogs are collaborative and have no single permanent owner. We create a second class of entity groups to hold the posts and metadata for each blog. A third class keys off the unique name claimed by each blog. The application relies on asynchronous messaging when a single user operation affects both blogs and profiles. For a lower-traffic operation like creating a new blog and claiming its unique name, two-phase commit is more convenient and performs adequately.

Maps Geographic data has no natural granularity of any consistent or convenient size. A mapping application can create entity groups by dividing the globe into non-overlapping patches. For mutations that span patches, the application uses two-phase commit to make them atomic. Patches must be large enough that two-phase transactions are uncommon, but small enough that each patch requires only a small write throughput. Unlike the previous examples, the number of entity groups does not grow with increased usage, so enough patches must be created initially for sufficient aggregate throughput at later scale.

Nearly all applications built on Megastore have found natural ways to draw entity group boundaries.

2.2.3 *Physical Layout*

We use Google's Bigtable [15] for scalable fault-tolerant storage within a single datacenter, allowing us to support arbitrary read and write throughput by spreading operations across multiple rows.

We minimize latency and maximize throughput by letting applications control the placement of data: through the selection of Bigtable instances and specification of locality within an instance.

To minimize latency, applications try to keep data near users and replicas near each other. They assign each entity group to the region or continent from which it is accessed most. Within that region they assign a triplet or quintuplet of replicas to datacenters with isolated failure domains.

For low latency, cache efficiency, and throughput, the data for an entity group are held in contiguous ranges of Bigtable rows. Our schema language lets applications control the placement of hierarchical data, storing data that is accessed together in nearby rows or denormalized into the same row.

3. A TOUR OF MEGASTORE

Megastore maps this architecture onto a feature set carefully chosen to encourage rapid development of scalable applications. This section motivates the tradeoffs and describes the developer-facing features that result.

3.1 API Design Philosophy

ACID transactions simplify reasoning about correctness, but it is equally important to be able to reason about performance. Megastore emphasizes *cost-transparent* APIs with runtime costs that match application developers' intuitions.

Normalized relational schemas rely on joins at query time to service user operations. This is not the right model for Megastore applications for several reasons:

- High-volume interactive workloads benefit more from predictable performance than from an expressive query language.

- Reads dominate writes in our target applications, so it pays to move work from read time to write time.
- Storing and querying hierarchical data is straightforward in key-value stores like Bigtable.

With this in mind, we designed a data model and schema language to offer fine-grained control over physical locality. Hierarchical layouts and declarative denormalization help eliminate the need for most joins. Queries specify scans or lookups against particular tables and indexes.

Joins, when required, are implemented in application code. We provide an implementation of the merge phase of the merge join algorithm, in which the user provides multiple queries that return primary keys for the same table in the same order; we then return the intersection of keys for all the provided queries.

We also have applications that implement outer joins with parallel queries. This typically involves an index lookup followed by parallel index lookups using the results of the initial lookup. We have found that when the secondary index lookups are done in parallel and the number of results from the first lookup is reasonably small, this provides an effective stand-in for SQL-style joins.

While schema changes require corresponding modifications to the query implementation code, this system guarantees that features are built with a clear understanding of their performance implications. For example, when users (who may not have a background in databases) find themselves writing something that resembles a nested-loop join algorithm, they quickly realize that it's better to add an index and follow the index-based join approach above.

3.2 Data Model

Megastore defines a data model that lies between the abstract tuples of an RDBMS and the concrete row-column storage of NoSQL. As in an RDBMS, the data model is declared in a *schema* and is strongly typed. Each schema has a set of *tables*, each containing a set of *entities*, which in turn contain a set of *properties*. Properties are named and typed values. The types can be strings, various flavors of numbers, or Google's Protocol Buffers [9]. They can be required, optional, or repeated (allowing a list of values in a single property). All entities in a table have the same set of allowable properties. A sequence of properties is used to form the primary key of the entity, and the primary keys must be unique within the table. Figure 3 shows an example schema for a simple photo storage application.

Megastore tables are either *entity group root* tables or *child* tables. Each child table must declare a single distinguished foreign key referencing a root table, illustrated by the `ENTITY GROUP KEY` annotation in Figure 3. Thus each child entity references a particular entity in its root table (called the *root entity*). An entity group consists of a root entity along with all entities in child tables that reference it. A Megastore instance can have several root tables, resulting in different classes of entity groups.

In the example schema of Figure 3, each user's photo collection is a separate entity group. The root entity is the `User`, and the `Photo`s are child entities. Note the `Photo.tag` field is repeated, allowing multiple tags per `Photo` without the need for a sub-table.

3.2.1 Pre-Joining with Keys

While traditional relational modeling recommends that

```
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
 required int64 user_id;
 required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
 required int64 user_id;
 required int32 photo_id;
 required int64 time;
 required string full_url;
 optional string thumbnail_url;
 repeated string tag;
} PRIMARY KEY(user_id, photo_id),
  IN TABLE User,
  ENTITY GROUP KEY(user_id) REFERENCES User;

CREATE LOCAL INDEX PhotosByTime
  ON Photo(user_id, time);

CREATE GLOBAL INDEX PhotosByTag
  ON Photo(tag) STORING (thumbnail_url);
```

Figure 3: Sample Schema for Photo Sharing Service

all primary keys take surrogate values, Megastore keys are chosen to cluster entities that will be read together. Each entity is mapped into a single Bigtable row; the primary key values are concatenated to form the Bigtable row key, and each remaining property occupies its own Bigtable column.

Note how the `Photo` and `User` tables in Figure 3 share a common `user_id` key prefix. The `IN TABLE User` directive instructs Megastore to colocate these two tables into the same Bigtable, and the key ordering ensures that `Photo` entities are stored adjacent to the corresponding `User`. This mechanism can be applied recursively to speed queries along arbitrary join depths. Thus, users can force hierarchical layout by manipulating the key order.

Schemas declare keys to be sorted ascending or descending, or to avert sorting altogether: the `SCATTER` attribute instructs Megastore to prepend a two-byte hash to each key. Encoding monotonically increasing keys this way prevents hotspots in large data sets that span Bigtable servers.

3.2.2 Indexes

Secondary indexes can be declared on any list of entity properties, as well as fields within protocol buffers. We distinguish between two high-level classes of indexes: *local* and *global* (see Figure 2). A local index is treated as separate indexes for each entity group. It is used to find data within an entity group. In Figure 3, `PhotosByTime` is an example of a local index. The index entries are stored in the entity group and are updated atomically and consistently with the primary entity data.

A global index spans entity groups. It is used to find entities without knowing in advance the entity groups that contain them. The `PhotosByTag` index in Figure 3 is global and enables discovery of photos marked with a given tag, regardless of owner. Global index scans can read data owned by many entity groups but are not guaranteed to reflect all recent updates.

Megastore offers additional indexing features:

3.2.2.1 Storing Clause.

Accessing entity data through indexes is normally a two-step process: first the index is read to find matching primary keys, then these keys are used to fetch entities. We provide a way to denormalize portions of entity data directly into index entries. By adding the `STORING` clause to an index, applications can store additional properties from the primary table for faster access at read time. For example, the `PhotosByTag` index stores the photo thumbnail URL for faster retrieval without the need for an additional lookup.

3.2.2.2 Repeated Indexes.

Megastore provides the ability to index repeated properties and protocol buffer sub-fields. Repeated indexes are a efficient alternative to child tables. `PhotosByTag` is a repeated index: each unique entry in the `tag` property causes one index entry to be created on behalf of the `Photo`.

3.2.2.3 Inline Indexes.

Inline indexes provide a way to denormalize data from source entities into a related target entity: index entries from the source entities appear as a virtual repeated column in the target entry. An inline index can be created on any table that has a foreign key referencing another table by using the first primary key of the target entity as the first components of the index, and physically locating the data in the same Bigtable as the target.

Inline indexes are useful for extracting slices of information from child entities and storing the data in the parent for fast access. Coupled with repeated indexes, they can also be used to implement many-to-many relationships more efficiently than by maintaining a many-to-many link table.

The `PhotosByTime` index could have been implemented as an inline index into the parent `User` table. This would make the data accessible as a normal index or as a virtual repeated property on `User`, with a time-ordered entry for each contained `Photo`.

3.2.3 Mapping to Bigtable

The Bigtable column name is a concatenation of the Megastore table name and the property name, allowing entities from different Megastore tables to be mapped into the same Bigtable row without collision. Figure 4 shows how data from the example photo application might look in Bigtable.

Within the Bigtable row for a root entity, we store the transaction and replication metadata for the entity group, including the transaction log. Storing all metadata in a single Bigtable row allows us to update it atomically through a single Bigtable transaction.

Each index entry is represented as a single Bigtable row; the row key of the cell is constructed using the indexed property values concatenated with the primary key of the indexed entity. For example, the `PhotosByTime` index row keys would be the tuple (*user_id*, *time*, *primary key*) for each photo. Indexing repeated fields produces one index entry per repeated element. For example, the primary key for a photo with three tags would appear in the `PhotosByTag` index thrice.

3.3 Transactions and Concurrency Control

Each Megastore entity group functions as a mini-database

Row key	User. name	Photo. time	Photo. tag	Photo. _url
101	John			
101,500		12:30:01	Dinner, Paris	...
101,502		12:15:22	Betty, Paris	...
102	Mary			

Figure 4: Sample Data Layout in Bigtable

that provides serializable ACID semantics. A transaction writes its mutations into the entity group's write-ahead log, then the mutations are applied to the data.

Bigtable provides the ability to store multiple values in the same row/column pair with different timestamps. We use this feature to implement multiversion concurrency control (MVCC): when mutations within a transaction are applied, the values are written at the timestamp of their transaction. Readers use the timestamp of the last fully applied transaction to avoid seeing partial updates. Readers and writers don't block each other, and reads are isolated from writes for the duration of a transaction.

Megastore provides *current*, *snapshot*, and *inconsistent* reads. Current and snapshot reads are always done within the scope of a single entity group. When starting a current read, the transaction system first ensures that all previously committed writes are applied; then the application reads at the timestamp of the latest committed transaction. For a snapshot read, the system picks up the timestamp of the last known fully applied transaction and reads from there, even if some committed transactions have not yet been applied. Megastore also provides inconsistent reads, which ignore the state of the log and read the latest values directly. This is useful for operations that have more aggressive latency requirements and can tolerate stale or partially applied data.

A write transaction always begins with a current read to determine the next available log position. The commit operation gathers mutations into a log entry, assigns it a timestamp higher than any previous one, and appends it to the log using Paxos. The protocol uses optimistic concurrency: though multiple writers might be attempting to write to the same log position, only one will win. The rest will notice the victorious write, abort, and retry their operations. Advisory locking is available to reduce the effects of contention. Batching writes through session affinity to a particular front-end server can avoid contention altogether.

The complete transaction lifecycle is as follows:

1. **Read**: Obtain the timestamp and log position of the last committed transaction.
2. **Application logic**: Read from Bigtable and gather writes into a log entry.
3. **Commit**: Use Paxos to achieve consensus for appending that entry to the log.
4. **Apply**: Write mutations to the entities and indexes in Bigtable.
5. **Clean up**: Delete data that is no longer required.

The write operation can return to the client at any point after Commit, though it makes a best-effort attempt to wait for the nearest replica to apply.

3.3.1 Queues

Queues provide transactional messaging between entity groups. They can be used for cross-group operations, to

batch multiple updates into a single transaction, or to defer work. A transaction on an entity group can atomically send or receive multiple messages in addition to updating its entities. Each message has a single sending and receiving entity group; if they differ, delivery is asynchronous. (See Figure 2.)

Queues offer a way to perform operations that affect many entity groups. For example, consider a calendar application in which each calendar has a distinct entity group, and we want to send an invitation to a group of calendars. A single transaction can atomically send invitation queue messages to many distinct calendars. Each calendar receiving the message will process the invitation in its own transaction which updates the invitee's state and deletes the message.

There is a long history of message queues in full-featured RDBMSs. Our support is notable for its scale: declaring a queue automatically creates an inbox on each entity group, giving us millions of endpoints.

3.3.2 Two-Phase Commit

Megastore supports two-phase commit for atomic updates across entity groups. Since these transactions have much higher latency and increase the risk of contention, we generally discourage applications from using the feature in favor of queues. Nevertheless, they can be useful in simplifying application code for unique secondary key enforcement.

3.4 Other Features

We have built a tight integration with Bigtable's full-text index in which updates and searches participate in Megastore's transactions and multiversion concurrency. A full-text index declared in a Megastore schema can index a table's text or other application-generated attributes.

Synchronous replication is sufficient defense against the most common corruptions and accidents, but backups can be invaluable in cases of programmer or operator error. Megastore's integrated backup system supports periodic full snapshots as well as incremental backup of transaction logs. The restore process can bring back an entity group's state to any point in time, optionally omitting selected log entries (as after accidental deletes). The backup system complies with legal and common sense principles for expiring deleted data.

Applications have the option of encrypting data at rest, including the transaction logs. Encryption uses a distinct key per entity group. We avoid granting the same operators access to both the encryption keys and the encrypted data.

4. REPLICATION

This section details the heart of our synchronous replication scheme: a low-latency implementation of Paxos. We discuss operational details and present some measurements of our production service.

4.1 Overview

Megastore's replication system provides a single, consistent view of the data stored in its underlying replicas. Reads and writes can be initiated from any replica, and ACID semantics are preserved regardless of what replica a client starts from. Replication is done per entity group by synchronously replicating the group's transaction log to a quorum of replicas. Writes typically require one round of inter-datacenter communication, and healthy-case reads run locally. Current reads have the following guarantees:

- A read always observes the last-acknowledged write.
- After a write has been observed, all future reads observe that write. (A write might be observed before it is acknowledged.)

4.2 Brief Summary of Paxos

The Paxos algorithm is a way to reach consensus among a group of replicas on a single value. It tolerates delayed or reordered messages and replicas that fail by stopping. A majority of replicas must be active and reachable for the algorithm to make progress—that is, it allows up to F faults with $2F + 1$ replicas. Once a value is *chosen* by a majority, all future attempts to read or write the value will reach the same outcome.

The ability to determine the outcome of a single value by itself is not of much use to a database. Databases typically use Paxos to replicate a transaction log, where a separate instance of Paxos is used for each position in the log. New values are written to the log at the position following the last chosen position.

The original Paxos algorithm [27] is ill-suited for high-latency network links because it demands multiple rounds of communication. Writes require at least two inter-replica roundtrips before consensus is achieved: a round of *prepares*, which reserves the right for a subsequent round of *accepts*. Reads require at least one round of prepares to determine the last chosen value. Real world systems built on Paxos reduce the number of roundtrips required to make it a practical algorithm. We will first review how master-based systems use Paxos, and then explain how we make Paxos efficient.

4.3 Master-Based Approaches

To minimize latency, many systems use a dedicated master to which all reads and writes are directed. The master participates in all writes, so its state is always up-to-date. It can serve reads of the current consensus state without any network communication. Writes are reduced to a single round of communication by piggybacking a prepare for the next write on each accept [14]. The master can batch writes together to improve throughput.

Reliance on a master limits flexibility for reading and writing. Transaction processing must be done near the master replica to avoid accumulating latency from sequential reads. Any potential master replica must have adequate resources for the system's full workload; slave replicas waste resources until the moment they become master. Master failover can require a complicated state machine, and a series of timers must elapse before service is restored. It is difficult to avoid user-visible outages.

4.4 Megastore's Approach

In this section we discuss the optimizations and innovations that make Paxos practical for our system.

4.4.1 Fast Reads

We set an early requirement that current reads should usually execute on any replica without inter-replica RPCs. Since writes usually succeed on all replicas, it was realistic to allow local reads everywhere. These *local reads* give us better utilization, low latencies in all regions, fine-grained read failover, and a simpler programming experience.

We designed a service called the *Coordinator*, with servers in each replica's datacenter. A coordinator server tracks a set of entity groups for which its replica has observed all Paxos writes. For entity groups in that set, the replica has sufficient state to serve local reads.

It is the responsibility of the write algorithm to keep coordinator state conservative. If a write fails on a replica's Bigtable, it cannot be considered committed until the group's key has been evicted from that replica's coordinator.

Since coordinators are simple, they respond more reliably and quickly than Bigtable. Handling of rare failure cases or network partitions is described in Section 4.7.

4.4.2 Fast Writes

To achieve fast single-roundtrip writes, Megastore adapts the pre-preparing optimization used by master-based approaches. In a master-based system, each successful write includes an implied prepare message granting the master the right to issue accept messages for the next log position. If the write succeeds, the prepares are honored, and the next write skips directly to the accept phase. Megastore does not use dedicated masters, but instead uses *leaders*.

We run an independent instance of the Paxos algorithm for each log position. The leader for each log position is a distinguished replica chosen alongside the preceding log position's consensus value. The leader arbitrates which value may use proposal number zero. The first writer to submit a value to the leader wins the right to ask all replicas to accept that value as proposal number zero. All other writers must fall back on two-phase Paxos.

Since a writer must communicate with the leader before submitting the value to other replicas, we minimize writer-leader latency. We designed our policy for selecting the next write's leader around the observation that most applications submit writes from the same region repeatedly. This leads to a simple but effective heuristic: use the closest replica.

4.4.3 Replica Types

So far all replicas have been *full* replicas, meaning they contain all the entity and index data and are able to service current reads. We also support the notion of a *witness* replica. Witnesses vote in Paxos rounds and store the write-ahead log, but do not apply the log and do not store entity data or indexes, so they have lower storage costs. They are effectively tie breakers and are used when there are not enough full replicas to form a quorum. Because they do not have a coordinator, they do not force an additional roundtrip when they fail to acknowledge a write.

Read-only replicas are the inverse of witnesses: they are non-voting replicas that contain full snapshots of the data. Reads at these replicas reflect a consistent view of some point in the recent past. For reads that can tolerate this staleness, read-only replicas help disseminate data over a wide geographic area without impacting write latency.

4.5 Architecture

Figure 5 shows the key components of Megastore for an instance with two full replicas and one witness replica.

Megastore is deployed through a client library and auxiliary servers. Applications link to the client library, which implements Paxos and other algorithms: selecting a replica for read, catching up a lagging replica, and so on.

Each application server has a designated *local replica*. The

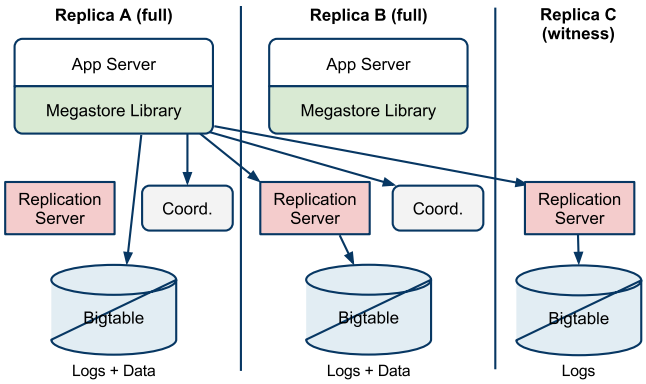


Figure 5: Megastore Architecture Example

client library makes Paxos operations on that replica durable by submitting transactions directly to the local Bigtable. To minimize wide-area roundtrips, the library submits remote Paxos operations to stateless intermediary *replication servers* communicating with their local Bigtables.

Client, network, or Bigtable failures may leave a write abandoned in an indeterminate state. Replication servers periodically scan for incomplete writes and propose no-op values via Paxos to bring them to completion.

4.6 Data Structures and Algorithms

This section details data structures and algorithms required to make the leap from consensus on a single value to a functioning replicated log.

4.6.1 Replicated Logs

Each replica stores mutations and metadata for the log entries known to the group. To ensure that a replica can participate in a write quorum even as it recovers from previous outages, we permit replicas to accept out-of-order proposals. We store log entries as independent cells in Bigtable.

We refer to a log replica as having "holes" when it contains an incomplete prefix of the log. Figure 6 demonstrates this scenario with some representative log replicas for a single Megastore entity group. Log positions 0-99 have been fully scavenged and position 100 is partially scavenged, because each replica has been informed that the other replicas will never request a copy. Log position 101 was accepted by all replicas. Log position 102 found a bare quorum in A and C. Position 103 is noteworthy for having been accepted by A and C, leaving B with a hole at 103. A conflicting write attempt has occurred at position 104 on replica A and B preventing consensus.

4.6.2 Reads

In preparation for a current read (as well as before a write), at least one replica must be brought up to date: all mutations previously committed to the log must be copied to and applied on that replica. We call this process *catchup*.

Omitting some deadline management, the algorithm for a current read (shown in Figure 7) is as follows:

1. **Query Local**: Query the local replica's coordinator to determine if the entity group is up-to-date locally.
2. **Find Position**: Determine the highest possibly-committed log position, and select a replica that has ap-

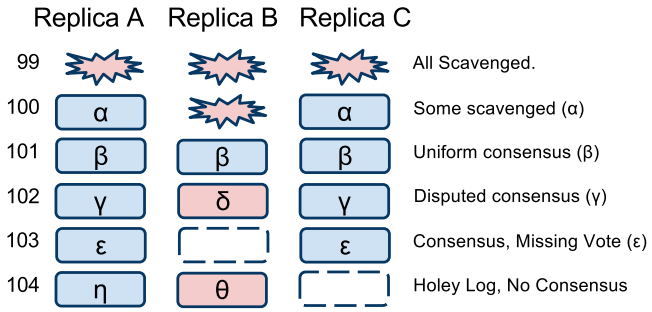


Figure 6: Write Ahead Log

plied through that log position.

(a) (*Local read*) If step 1 indicates that the local replica is up-to-date, read the highest accepted log position and timestamp from the local replica.

(b) (*Majority read*) If the local replica is not up-to-date (or if step 1 or step 2a times out), read from a majority of replicas to find the maximum log position that any replica has seen, and pick a replica to read from. We select the most responsive or up-to-date replica, not always the local replica.

3. **Catchup**: As soon as a replica is selected, catch it up to the maximum known log position as follows:

   (a) For each log position in which the selected replica does not know the consensus value, read the value from another replica. For any log positions without a known-committed value available, invoke Paxos to propose a no-op write. Paxos will drive a majority of replicas to converge on a single value—either the no-op or a previously proposed write.

   (b) Sequentially apply the consensus value of all unapplied log positions to advance the replica's state to the distributed consensus state.

   In the event of failure, retry on another replica.

4. **Validate**: If the local replica was selected and was not previously up-to-date, send the coordinator a *validate* message asserting that the (*entity group*, *replica*) pair reflects all committed writes. Do not wait for a reply—if the request fails, the next read will retry.

5. **Query Data**: Read the selected replica using the timestamp of the selected log position. If the selected replica becomes unavailable, pick an alternate replica, perform catchup, and read from it instead. The results of a single large query may be assembled transparently from multiple replicas.

Note that in practice 1 and 2a are executed in parallel.

4.6.3 Writes

Having completed the read algorithm, Megastore observes the next unused log position, the timestamp of the last write, and the next leader replica. At commit time all pending changes to the state are packaged and proposed, with a timestamp and next leader nominee, as the consensus value for the next log position. If this value wins the distributed

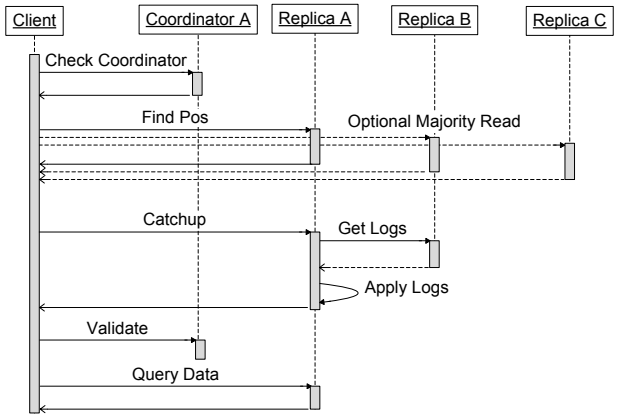


Figure 7: Timeline for reads with local replica A

consensus, it is applied to the state at all full replicas; otherwise the entire transaction is aborted and must be retried from the beginning of the read phase.

As described above, coordinators keep track of the entity groups that are up-to-date in their replica. If a write is not accepted on a replica, we must remove the entity group's key from that replica's coordinator. This process is called *invalidation*. Before a write is considered committed and ready to apply, all full replicas must have accepted or had their coordinator invalidated for that entity group.

The write algorithm (shown in Figure 8) is as follows:

1. **Accept Leader**: Ask the leader to accept the value as proposal number zero. If successful, skip to step 3.
2. **Prepare**: Run the Paxos Prepare phase at all replicas with a higher proposal number than any seen so far at this log position. Replace the value being written with the highest-numbered proposal discovered, if any.
3. **Accept**: Ask remaining replicas to accept the value. If this fails on a majority of replicas, return to step 2 after a randomized backoff.
4. **Invalidate**: Invalidate the coordinator at all full replicas that did not accept the value. Fault handling at this step is described in Section 4.7 below.
5. **Apply**: Apply the value's mutations at as many replicas as possible. If the chosen value differs from that originally proposed, return a conflict error.

Step 1 implements the "fast writes" of Section 4.4.2. Writers using single-phase Paxos skip Prepare messages by sending an Accept command at proposal number zero. The next leader replica selected at log position n arbitrates the value used for proposal zero at $n + 1$. Since multiple proposers may submit values with proposal number zero, serializing at this replica ensures only one value corresponds with that proposal number for a particular log position.

In a traditional database system, the *commit point* (when the change is durable) is the same as the *visibility point* (when reads can see a change and when a writer can be notified of success). In our write algorithm, the commit point is after step 3 when the write has won the Paxos round, but the visibility point is after step 4. Only after all full replicas have accepted or had their coordinators invalidated can the write be acknowledged and the changes applied. Acknowledging before step 4 could violate our consistency guarantees: a current read at a replica whose invalidation was skipped might

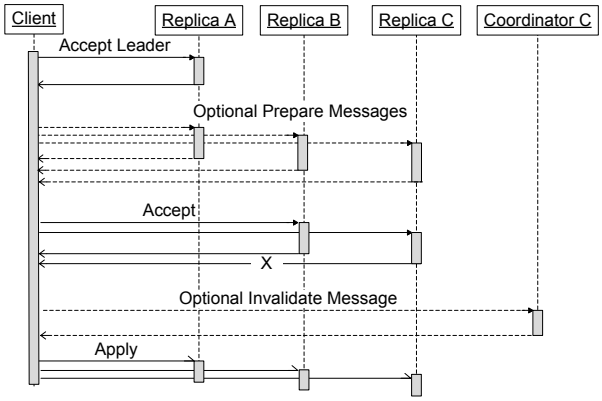


Figure 8: Timeline for writes

fail to observe the acknowledged write.

4.7 Coordinator Availability

Coordinator processes run in each datacenter and keep state only about their local replica. In the write algorithm above, each full replica must either accept or have its coordinator invalidated, so it might appear that any single replica failure (Bigtable and coordinator) will cause unavailability.

In practice this is not a common problem. The coordinator is a simple process with no external dependencies and no persistent storage, so it tends to be much more stable than a Bigtable server. Nevertheless, network and host failures can still make the coordinator unavailable.

4.7.1 Failure Detection

To address network partitions, coordinators use an out-of-band protocol to identify when other coordinators are up, healthy, and generally reachable.

We use Google's Chubby lock service [13]: coordinators obtain specific Chubby locks in remote datacenters at startup. To process requests, a coordinator must hold a majority of its locks. If it ever loses a majority of its locks from a crash or network partition, it will revert its state to a conservative default, considering all entity groups in its purview to be out-of-date. Subsequent reads at the replica must query the log position from a majority of replicas until the locks are regained and its coordinator entries are revalidated.

Writers are insulated from coordinator failure by testing whether a coordinator has lost its locks: in that scenario, a writer knows that the coordinator will consider itself invalidated upon regaining them.

This algorithm risks a brief (tens of seconds) write outage when a datacenter containing live coordinators suddenly becomes unavailable—all writers must wait for the coordinator's Chubby locks to expire before writes can complete (much like waiting for a master failover to trigger). Unlike after a master failover, reads and writes can proceed smoothly while the coordinator's state is reconstructed. This brief and rare outage risk is more than justified by the steady state of fast local reads it allows.

The coordinator liveness protocol is vulnerable to asymmetric network partitions. If a coordinator can maintain the leases on its Chubby locks, but has otherwise lost contact with proposers, then affected entity groups will experience a write outage. In this scenario an operator performs a manual step to disable the partially isolated coordinator. We have faced this condition only a handful of times.

4.7.2 Validation Races

In addition to availability issues, protocols for reading and writing to the coordinator must contend with a variety of race conditions. Invalidate messages are always safe, but validate messages must be handled with care. Races between validates for earlier writes and invalidates for later writes are protected in the coordinator by always sending the log position associated with the action. Higher numbered invalidates always trump lower numbered validates. There are also races associated with a crash between an invalidate by a writer at position n and a validate at some position $m < n$. We detect crashes using a unique epoch number for each incarnation of the coordinator: validates are only allowed to modify the coordinator state if the epoch remains unchanged since the most recent read of the coordinator.

In summary, using coordinators to allow fast local reads from any datacenter is not free in terms of the impact to availability. But in practice most of the problems with running the coordinator are mitigated by the following factors:

- Coordinators are much simpler processes than Bigtable servers, have many fewer dependencies, and are thus naturally more available.
- Coordinators' simple, homogeneous workload makes them cheap and predictable to provision.
- Coordinators' light network traffic allows using a high network QoS with reliable connectivity.
- Operators can centrally disable coordinators for maintenance or unhealthy periods. This is automatic for certain monitoring signals.
- A quorum of Chubby locks detects most network partitions and node unavailability.

4.8 Write Throughput

Our implementation of Paxos has interesting tradeoffs in system behavior. Application servers in multiple datacenters may initiate writes to the same entity group and log position simultaneously. All but one of them will fail and need to retry their transactions. The increased latency imposed by synchronous replication increases the likelihood of conflicts for a given per-entity-group commit rate.

Limiting that rate to a few writes per second per entity group yields insignificant conflict rates. For apps whose entities are manipulated by a small number of users at a time, this limitation is generally not a concern. Most of our target customers scale write throughput by sharding entity groups more finely or by ensuring replicas are placed in the same region, decreasing both latency and conflict rate.

Applications with some server "stickiness" are well positioned to batch user operations into fewer Megastore transactions. Bulk processing of Megastore queue messages is a common batching technique, reducing the conflict rate and increasing aggregate throughput.

For groups that must regularly exceed a few writes per second, applications can use the fine-grained advisory locks dispensed by coordinator servers. Sequencing transactions back-to-back avoids the delays associated with retries and the reversion to two-phase Paxos when a conflict is detected.

4.9 Operational Issues

When a particular full replica becomes unreliable or loses connectivity, Megastore's performance can degrade. We have

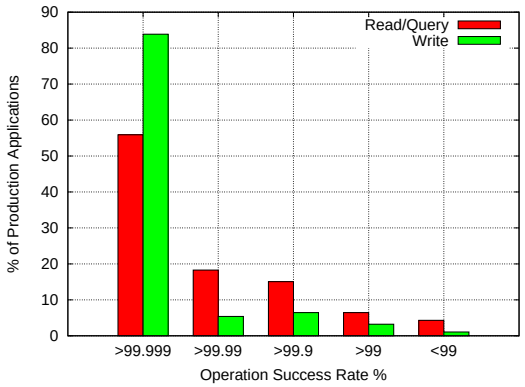


Figure 9: Distribution of Availability

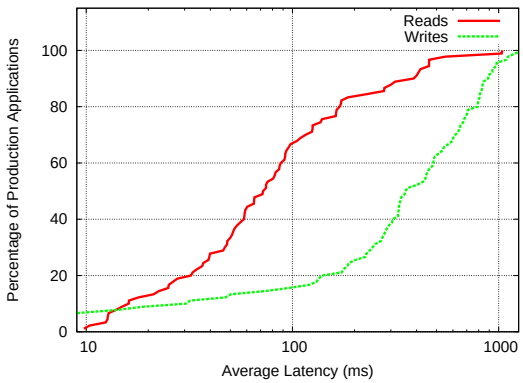


Figure 10: Distribution of Average Latencies

a number of ways to respond to these failures, including: routing users away from the problematic replica, disabling its coordinators, or disabling it entirely. In practice we rely on a combination of techniques, each with its own tradeoffs.

The first and most important response to an outage is to disable Megastore clients at the affected replica by rerouting traffic to application servers near other replicas. These clients typically experience the same outage impacting the storage stack below them, and might be unreachable from the outside world.

Rerouting traffic alone is insufficient if unhealthy coordinator servers might continue to hold their Chubby locks. The next response is to disable the replica's coordinators, ensuring that the problem has a minimal impact on write latency. (Section 4.7 described this process in more detail.) Once writers are absolved of invalidating the replica's coordinators, an unhealthy replica's impact on write latency is limited. Only the initial "accept leader" step in the write algorithm depends on the replica, and we maintain a tight deadline before falling back on two-phase Paxos and nominating a healthier leader for the next write.

A more draconian and rarely used action is to disable the replica entirely: neither clients nor replication servers will attempt to communicate with it. While sequestering the replica can seem appealing, the primary impact is a hit to availability: one less replica is eligible to help writers form a quorum. The valid use case is when attempted operations might cause harm—e.g. when the underlying Bigtable is severely overloaded.

4.10 Production Metrics

Megastore has been deployed within Google for several years; more than 100 production applications use it as their storage service. In this section, we report some measurements of its scale, availability, and performance.

Figure 9 shows the distribution of availability, measured on a per-application, per-operation basis. Most of our customers see extremely high levels of availability (at least five nines) despite a steady stream of machine failures, network hiccups, datacenter outages, and other faults. The bottom end of our sample includes some pre-production applications that are still being tested and batch processing applications with higher failure tolerances.

Average read latencies are tens of milliseconds, depending on the amount of data, showing that most reads are local. Most users see average write latencies of 100–400 milliseconds, depending on the distance between datacenters, the size of the data being written, and the number of full replicas. Figure 10 shows the distribution of average latency for read and commit operations.

5. EXPERIENCE

Development of the system was aided by a strong emphasis on testability. The code is instrumented with numerous (but cheap) assertions and logging, and has thorough unit test coverage. But the most effective bug-finding tool was our network simulator: the *pseudo-random test* framework. It is capable of exploring the space of all possible orderings and delays of communications between simulated nodes or threads, and deterministically reproducing the same behavior given the same seed. Bugs were exposed by finding a problematic sequence of events triggering an assertion failure (or incorrect result), often with enough log and trace information to diagnose the problem, which was then added to the suite of unit tests. While an exhaustive search of the scheduling state space is impossible, the pseudo-random simulation explores more than is practical by other means. Through running thousands of simulated hours of operation each night, the tests have found many surprising problems.

In real-world deployments we have observed the expected performance: our replication protocol optimizations indeed provide local reads most of the time, and writes with about the overhead of a single WAN roundtrip. Most applications have found the latency tolerable. Some applications are designed to hide write latency from users, and a few must choose entity group boundaries carefully to maximize their write throughput. This effort yields major operational advantages: Megastore's latency tail is significantly shorter than that of the underlying layers, and most applications can withstand planned and unplanned outages with little or no manual intervention.

Most applications use the Megastore schema language to model their data. Some have implemented their own *entity-attribute-value* model within the Megastore schema language, then used their own application logic to model their data (most notably, Google App Engine [8]). Some use a hybrid of the two approaches. Having the dynamic schema built on top of the static schema, rather than the other way around, allows most applications to enjoy the performance, usability,

and integrity benefits of the static schema, while still giving the option of a dynamic schema to those who need it.

The term "high availability" usually signifies the ability to mask faults to make a collection of systems more reliable than the individual systems. While fault tolerance is a highly desired goal, it comes with it its own pitfalls: it often hides persistent underlying problems. We have a saying in the group: "Fault tolerance is fault masking". Too often, the resilience of our system coupled with insufficient vigilance in tracking the underlying faults leads to unexpected problems: small transient errors on top of persistent uncorrected problems cause significantly larger problems.

Another issue is flow control. An algorithm that tolerates faulty participants can be heedless of slow ones. Ideally a collection of disparate machines would make progress only as fast as the least capable member. If slowness is interpreted as a fault, and tolerated, the fastest majority of machines will process requests at their own pace, reaching equilibrium only when slowed down by the load of the laggers struggling to catch up. We call this anomaly *chain gang throttling*, evoking the image of a group of escaping convicts making progress only as quickly as they can drag the stragglers.

A benefit of Megastore's write-ahead log has been the ease of integrating external systems. Any idempotent operation can be made a step in applying a log entry.

Achieving good performance for more complex queries requires attention to the physical data layout in Bigtable. When queries are slow, developers need to examine Bigtable traces to understand why their query performs below their expectations. Megastore does not enforce specific policies on block sizes, compression, table splitting, locality group, nor other tuning controls provided by Bigtable. Instead, we expose these controls, providing application developers with the ability (and burden) of optimizing performance.

6. RELATED WORK

Recently, there has been increasing interest in NoSQL data storage systems to meet the demand of large web applications. Representative work includes Bigtable [15], Cassandra [6], and Yahoo PNUTS [16]. In these systems, scalability is achieved by sacrificing one or more properties of traditional RDBMS systems, e.g., transactions, schema support, query capability [12, 33]. These systems often reduce the scope of transactions to the granularity of single key access and thus place a significant hurdle to building applications [18, 32]. Some systems extend the scope of transactions to multiple rows within a single table, for example the Amazon SimpleDB [5] uses the concept of *domain* as the transactional unit. Yet such efforts are still limited because transactions cannot cross tables or scale arbitrarily. Moreover, most current scalable data storage systems lack the rich data model of an RDBMS, which increases the burden on developers. Combining the merits from both database and scalable data stores, Megastore provides transactional ACID guarantees within a entity group and provides a flexible data model with user-defined schema, database-style and full-text indexes, and queues.

Data replication across geographically distributed datacenters is an indispensable means of improving availability in state-of-the-art storage systems. Most prevailing data storage systems use asynchronous replication schemes with a weaker consistency model. For example, Cassandra [6], HBase [1], CouchDB [7], and Dynamo [19] use an eventual consistency model and PNUTS uses "timeline" consistency [16]. By comparison, synchronous replication guarantees strong transactional semantics over wide-area networks and improves the performance of current reads.

Synchronous replication for traditional RDBMS systems presents a performance challenge and is difficult to scale [21]. Some proposed workarounds allow for strong consistency via asynchronous replication. One approach lets updates complete before their effects are replicated, passing the synchronization delay on to transactions that need to read the updated state [26]. Another approach routes writes to a single master while distributing read-only transactions among a set of replicas [29]. The updates are asynchronously propagated to the remaining replicas, and reads are either delayed or sent to replicas that have already been synchronized. A recent proposal for efficient synchronous replication introduces an ordering preprocessor that schedules incoming transactions deterministically, so that they can be independently applied at multiple replicas with identical results [31]. The synchronization burden is shifted to the preprocessor, which itself would have to be made scalable.

Until recently, few have used Paxos to achieve synchronous replication. SCALARIS is one example that uses the Paxos commit protocol [22] to implement replication for a distributed hash table [30]. Keyspace [2] also uses Paxos to implement replication on a generic key-value store. However the scalability and performance of these systems is not publicly known. Megastore is perhaps the first large-scale storage systems to implement Paxos-based replication across datacenters while satisfying the scalability and performance requirements of scalable web applications in the cloud.

Conventional database systems provide mature and sophisticated data management features, but have difficulties in serving large-scale interactive services targeted by this paper [33]. Open source database systems such as MySQL [10] do not scale up to the levels we require [17], while expensive commercial database systems like Oracle [4] significantly increase the total cost of ownership in large deployments in the cloud. Furthermore, neither of them offer fault-tolerant synchronous replication mechanism [3, 11], which is a key piece to build interactive services in the cloud.

7. CONCLUSION

In this paper we present Megastore, a scalable, highly available datastore designed to meet the storage requirements of interactive Internet services. We use Paxos for synchronous wide area replication, providing lightweight and fast failover of individual operations. The latency penalty of synchronous replication across widely distributed replicas is more than offset by the convenience of a single system image and the operational benefits of carrier-grade availability. We use Bigtable as our scalable datastore while adding richer primitives such as ACID transactions, indexes, and queues. Partitioning the database into entity group sub-databases provides familiar transactional features for most operations while allowing scalability of storage and throughput.

Megastore has over 100 applications in production, facing both internal and external users, and providing infrastructure for higher levels. The number and diversity of these applications is evidence of Megastore's ease of use, generality, and power. We hope that Megastore demonstrates the viability of a middle ground in feature set and replication consistency for today's scalable storage systems.

8. ACKNOWLEDGMENTS

Steve Newman, Jonas Karlsson, Philip Zeyliger, Alex Dingle, and Peter Stout all made substantial contributions to Megastore. We also thank Tushar Chandra, Mike Burrows, and the Bigtable team for technical advice, and Hector Gonzales, Jayant Madhavan, Ruth Wang, and Kavita Guliani for assistance with the paper. Special thanks to Adi Ofer for providing the spark to make this paper happen.

9. REFERENCES

[1] Apache HBase. http://hbase.apache.org/, 2008.

[2] Keyspace: A consistently replicated, highly-available key-value store. http://scalien.com/whitepapers/.

[3] MySQL Cluster. http://dev.mysql.com/tech-resources/articles/mysql_clustering_ch5.html, 2010.

[4] Oracle Database. http://www.oracle.com/us/-products/database/index.html, 2007.

[5] Amazon SimpleDB. http://aws.amazon.com/simpledb/, 2007.

[6] Apache Cassandra. http://incubator.apache.org/cassandra/, 2008.

[7] Apache CouchDB. http://couchdb.apache.org/, 2008.

[8] Google App Engine. http://code.google.com/appengine/, 2008.

[9] Google Protocol Buffers: Google's data interchange format. http://code.google.com/p/protobuf/, 2008.

[10] MySQL. http://www.mysql.com, 2009.

[11] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the performance of wide-area synchronous database replication. Technical Report CNDS-2002-4, Johns Hopkins University, 2002.

[12] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, New York, NY, USA, 2010. ACM.

[18] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, New York, NY, USA, 2010. ACM.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[20] J. Furman, J. S. Karlsson, J.-M. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A scalable data system for user facing applications. In *ACM SIGMOD/PODS Conference*, 2008.

[21] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[22] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[23] S. Gustavsson and S. F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 105–107, New York, NY, USA, 2002. ACM.

[24] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[25] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[26] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly consistent replication for a bargain. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 52 –63, 2010.

[27] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[28] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, Microsoft Research, 2009.

[29] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[30] F. Schintke, A. Reinefeld, S. e. Haridi, and T. Schutt. Enhanced paxos commit for transactions on dhts. In *10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 448–454, 2010.

[31] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[32] S. Wu, D. Jiang, B. C. Ooi, and K. L. W. Towards elastic transactional cloud storage with range query support. In *Int'l Conference on Very Large Data Bases (VLDB)*, 2010.

[33] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.

Relational Cloud: A Database-as-a-Service for the Cloud

Carlo Curino
curino@mit.edu

Evan P. C. Jones
evanj@mit.edu

Raluca Ada Popa
ralucap@mit.edu

Nirmesh Malviya
nirmesh@csail.mit.edu

Eugene Wu
eugenewu@mit.edu

Sam Madden
madden@csail.mit.edu

Hari Balakrishnan
hari@csail.mit.edu

Nickolai Zeldovich
nickolai@csail.mit.edu

ABSTRACT

This paper introduces a new transactional "database-as-a-service" (DBaaS) called **Relational Cloud**. A DBaaS promises to move much of the operational burden of provisioning, configuration, scaling, performance tuning, backup, privacy, and access control from the database users to the service operator, offering lower overall costs to users. Early DBaaS efforts include Amazon RDS and Microsoft SQL Azure, which are promising in terms of establishing the market need for such a service, but which do not address three important challenges: *efficient multi-tenancy*, *elastic scalability*, and *database privacy*. We argue that these three challenges must be overcome before outsourcing database software and management becomes attractive to many users, and cost-effective for service providers. The key technical features of Relational Cloud include: (1) a workload-aware approach to multi-tenancy that identifies the workloads that can be co-located on a database server, achieving higher consolidation and better performance than existing approaches; (2) the use of a graph-based data partitioning algorithm to achieve near-linear elastic scale-out even for complex transactional workloads; and (3) an adjustable security scheme that enables SQL queries to run over encrypted data, including ordering operations, aggregates, and joins. An underlying theme in the design of the components of Relational Cloud is the notion of *workload awareness*: by monitoring query patterns and data accesses, the system obtains information useful for various optimization and security functions, reducing the configuration effort for users and operators.

1. INTRODUCTION

Relational database management systems (DBMSs) are an integral and indispensable component in most computing environments today, and their importance is unlikely to diminish. With the advent of hosted cloud computing and storage, the opportunity to offer a DBMS as an outsourced service is gaining momentum, as witnessed by Amazon's RDS and Microsoft's SQL Azure (see §7). Such a **database-as-a-service (DBaaS)** is attractive for two reasons. First, due to economies of scale, the hardware and energy costs incurred by users are likely to be much lower when they are paying for a share of a service rather than running everything themselves. Second, the costs incurred in a well-designed DBaaS will be proportional to actual usage ("pay-per-use")—this applies to both software licensing and administrative costs. The latter are often a significant expense because of the specialized expertise required to extract good performance from commodity DBMSs. By centralizing and automating many database management tasks, a DBaaS can substantially reduce operational costs *and* perform well.

From the viewpoint of the operator of a DBaaS, by taking advantage of the lack of correlation between workloads of different applications, the service can be run using far fewer machines than if each workload was independently provisioned for its peak.

This paper describes the challenges and requirements of a large-scale, multi-node DBaaS, and presents the design principles and implementation status of Relational Cloud, a DBaaS we are building at MIT (see `http://relationalcloud.com`). Relational Cloud is appropriate for a single organization with many individual databases deployed in a "private" cloud, or as a service offered via "public" cloud infrastructure to multiple organizations. In both cases, our vision is that users should have access to all the features of a SQL relational DBMS, without worrying about provisioning the hardware resources, configuring software, achieving desired security, providing access control and data privacy, and tuning performance. All these functions are outsourced to the DBaaS.

There are three challenges that drive the design of Relational Cloud: efficient multi-tenancy to minimize the hardware footprint required for a given (or predicted) workload, elastic scale-out to handle growing workloads, and database privacy.

Efficient multi-tenancy. Given a set of databases and workloads, what is the best way to serve them from a given set of machines? The goal is to minimize the number of machines required, while meeting application-level query performance goals. To achieve this, our system must understand the resource requirements of individual workloads, how they combine when co-located on one machine, and how to take advantage of the temporal variations of each workload to maximize hardware utilization while avoiding overcommitment.

One approach to this problem would be to use virtual machines (VMs); a typical design would pack each individual DB instance into a VM and multiple VMs on a single physical machine. However, our experiments show that such a "DB-in-VM" approach requires $2\times$ to $3\times$ more machines to consolidate the same number of workloads and that for a fixed level of consolidation delivers $6\times$ to $12\times$ less performance than the approach we advocate. The reason is that each VM contains a separate copy of the OS and database, and each database has its own buffer pool, forces its own log to disk, etc. Instead, our approach uses a single database server on each machine, which hosts multiple logical databases. Relational Cloud periodically determines which databases should be placed on which machines using a novel non-linear optimization formulation, combined with a cost model that estimates the combined resource utilization of multiple databases running on a machine. The design of Relational Cloud also includes a lightweight mechanism to perform *live migration* of databases between machines.

Elastic scalability. A good DBaaS must support database and workloads of different sizes. The challenge arise when a database workload exceeds the capacity of a single machine. A DBaaS must therefore support *scale-out*, where the responsibility for query processing (and the corresponding data) is *partitioned* amongst multiple nodes to achieve higher throughput. But what is the best way

to partition databases for scale-out? The answer depends on the way in which transactions and data items relate to one another. In Relational Cloud, we use a recently developed *workload-aware partitioner* [5], which uses graph partitioning to automatically analyze complex query workloads and map data items to nodes to minimize the number of multi-node transactions/statements. Statements and transactions spanning multiple nodes incur significant overhead, and are the main limiting factor to linear scalability in practice. Our approach makes few assumptions on the data or queries, and works well even for skewed workloads or when the data exhibits complex many-to-many relationships.

Privacy. A significant barrier to deploying databases in the cloud is the perceived lack of privacy, which in turn reduces the degree of trust users are willing to place in the system. If clients were to encrypt all the data stored in the DBaaS, then the privacy concerns would largely be eliminated. The question then is, how can the DBaaS execute queries over the encrypted data? In Relational Cloud, we have developed *CryptDB*, a set of techniques designed to provide privacy (e.g., to prevent administrators from seeing a user's data) with an acceptable impact on performance (only a 22.5% reduction in throughput on TPC-C in our preliminary experiments). Database administrators can continue to manage and tune the databases, and users are guaranteed data privacy. The key notion is that of *adjustable security*: CryptDB employs different encryption levels for different types of data, based on the types of queries that users run. Queries are evaluated on the encrypted data, and sent back to the client for final decryption; no query processing runs on the client.

A unifying theme in our approach to these three big challenges is *workload-awareness*. Our main design principle is to monitor the actual query patterns and data accesses, and then employ mechanisms that use these observations to perform various optimization and security functions.

2. SYSTEM DESIGN

Relational Cloud uses existing unmodified DBMS engines as the back-end query processing and storage nodes. Each back-end node runs a single *database server*. The set of back-end machines can change dynamically in response to load. Each *tenant* of the system—which we define as a billable entity (a distinct user with a set of applications, a business unit, or a company)—can load one or more databases. A database has one or more tables, and an associated *workload*, defined as the set of queries and transactions issued to it (the set may not be known until run-time). Relational Cloud does *not* mix the data of two different tenants into a common database or table (unlike [1]), but databases belonging to different tenants will usually run within the same database server.

Applications communicate with Relational Cloud using a standard connectivity layer such as JDBC. They communicate with the Relational Cloud front-end using a special driver that ensures their data is kept private (e.g., cannot be read by the database administrator)—this is described in more detail below. When the front-end receives SQL statements from clients, it consults the *router*, which analyzes each SQL statement and uses its metadata to determine the execution nodes and plan. The front-end coordinates multi-node transactions, produces a distributed execution plan, and handles fail-over. It also provides a degree of performance isolation by controlling the rate at which queries from different tenants are dispatched.

The front-end monitors the access patterns induced by the workloads and the load on the database servers. Relational Cloud uses this information to periodically determine the best way to: (1) *partition* each database into one or more pieces, producing multiple partitions when the load on a database exceeds the capacity of a single machine (§3), (2) *place* the database partitions on the back-end machines to both minimize the number of machines and balance load, *migrate* the partitions as needed without causing downtime, and *replicate* the data for availability (§4), and (3) *secure* the data and process the queries so that they can run on untrusted back-ends over encrypted data (§5). The Relational Cloud system architecture is shown in Figure 1, which depicts these functions and demarcates the trusted and untrusted regions.

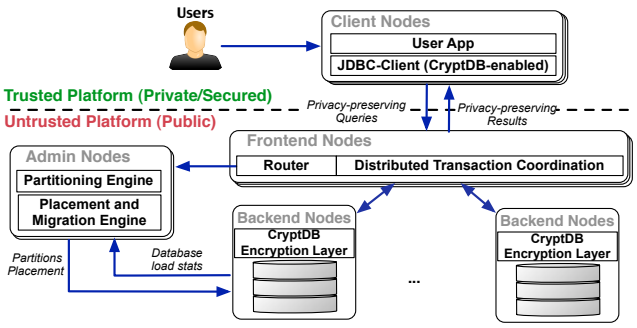


Figure 1: Relational Cloud Architecture.

Applications communicate with the Relational Cloud front-end using a CryptDB-enabled driver on the client, which encrypts and decrypts user data and rewrites queries to guarantee privacy. On the back-end nodes, CryptDB exploits a combination of server-side cryptography and user-defined functions (UDFs) to enable efficient SQL processing—particularly ordering, aggregates, and joins, which are all more challenging than simple selections and projections—over encrypted data.

Current status: We have developed the various components of Relational Cloud and are in the process of integrating them into a single coherent system, prior to offering it as a service on a public cloud. We have implemented the distributed transaction coordinator along with the routing, partitioning, replication, and CryptDB components. Our transaction coordinator supports both MySQL and Postgres back-ends, and have implemented a JDBC public interface. Given a query trace, we can analyze and automatically generate a good partitioning for it, and then run distributed transactions against those partitions. The transaction coordinator supports active fail-over to replicas in the event of a failure. We have developed a placement and migration engine that monitors database server statistics, OS statistics, and hardware loads, and uses historic statistics to predict the combined load placed by multiple workloads. It uses a non-linear, integer programming solver to optimally allocate partitions to servers. We are currently implementing live migration.

Other papers (either published [5] or in preparation) detail the individual components, which are of independent interest. This paper focuses on the Relational Cloud system and on how the components address the challenges of running a large-scale DBaaS, rather than on the more mundane engineering details of the DBaaS implementation or on the detailed design and performance of the components. (That said, in later sections we present the key performance results for the main components of Relational Cloud to show that the integrated system is close to being operational.) At the CIDR conference, we propose to demonstrate Relational Cloud.

3. DATABASE PARTITIONING

Relational Cloud uses database partitioning for two purposes: (1) to scale a single database to multiple nodes, useful when the load exceeds the capacity of a single machine, and (2) to enable more granular placement and load balance on the back-end machines compared to placing entire databases.

The current partitioning strategy is well-suited to OLTP and Web workloads, but the principles generalize to other workloads as well (such as OLAP). OLTP/Web workloads are characterized by short-lived transactions/queries with little internal parallelism. The way to scale these workloads is to partition the data in a way that minimizes the number of multi-node transactions (i.e., most transactions should complete by touching data on only one node), and then place the different partitions on different nodes. The goal is to minimize the number of cross-node distributed transactions, which incur overhead both because of the extra work done on each node and because of the increase in the time spent holding locks at the back-ends.

Relational Cloud uses a workload-aware partitioning strategy. The front-end has a component that periodically analyzes query execution traces to identify sets of tuples that are accessed together within individual transactions. The algorithm represents the execution trace as a graph. Each node represents a tuple (or collection of tuples) and an edge is drawn between any two nodes whose tuples are touched within a single transaction. The weight on an edge reflects how often such pair-wise accesses occur in a workload. Relational Cloud uses graph partitioning [13] to find ℓ balanced logical partitions, while minimizing the total weight of the cut edges. This minimization corresponds to find a partitioning of the database tuples that minimizes the number of distributed transactions.

The output of the partitioner is an assignment of individual tuples to logical partitions. Relational Cloud now has to come up with a succinct representation of these partitions, because the front-end's router needs a compact way to determine where to dispatch a given SQL statement. Relational Cloud solves this problem by finding a set of predicates on the tuple attributes. It is natural to formulate this problem as a classification problem, where we are given a set of tuples (the tuple attributes are features), and a partition label for each tuple (the classification attribute). The system extracts a set of candidate attributes from the predicates used in the trace. The attribute values are fed into a decision tree algorithm together with the partitioning labels. If the decision tree successfully generalizes the partitioning with few simple predicates, a good *explanation* for the graph partitioning is found. If no predicate-based explanation is found (e.g., if thousands of predicates are generated), the system falls back to lookup tables to represent the partitioning scheme.

The strength of this approach is its independence from schema layout and foreign key information, which allows it to discover intrinsic correlations hidden in the data. As a consequence, this approach is effective in partitioning databases containing multiple many-to-many relationships—typical in social-network scenarios—and in handling skewed workloads [5].

The main practical difficulty we encountered was in scaling the graph representation. The naïve approach leads to a graph with N nodes and up to N^2 edges for an N-tuple database, which is untenable because existing graph partitioning implementations scale only to a few tens of millions of nodes. For this reason, we devised a series of heuristics that effectively limit the size of the graph. The two most useful heuristics used in Relational Cloud are: (1) *blanket statement removal*, i.e., the exclusion from the graph occasional statements that scan large portions of the database and (2) *sampling* tuples and transactions.

4. PLACEMENT AND MIGRATION

Resource allocation is a major challenge when designing a scalable, multi-tenant service like Relational Cloud. Problems include: (i) monitoring the resource requirements of each workload, (ii) predicting the load multiple workloads will generate when run together on a server, (iii) assigning workloads to physical servers, and (iv) migrating them between physical nodes.

In Relational Cloud, a new database and workload are placed arbitrarily on some set of nodes for applications, while at the same time set up in a staging area where they run on dedicated hardware. During this time, the system monitors their resource consumption in the staging area (and for the live version). The resulting time-dependent resource profile is used to predict how this workload will interact with the others currently running in the service, and whether the workload needs to be partitioned. If a workload needs to be partitioned, it is split using the algorithms described in the previous section. After that, an allocation algorithm is run to place the each workload or partition onto existing servers, together with other partitions and workloads.

We call the monitoring and consolidation engine we developed for this purpose *Kairos*; a complete paper on Kairos is currently under submission. It takes as input an existing (non-consolidated) collection of workloads, and a set of target physical machines on which to consolidate those workload, and performs the aforementioned analysis and placement tasks. Its key components are:

1. *Resource Monitor*: Through an automated statistics collection process, the resource monitor captures a number of DBMS and OS statistics from a running database. One monitoring challenge is estimating the RAM required by a workload, since a standalone DBMS will tend to fill the entire buffer pool with pages, even if many of those pages aren't actively in use. To precisely measure working set size, Kairos slowly grows and repeatedly accesses a temporary probe table, while monitoring amount of disk activity on the system. Once the probe table begins to evict tuples in the working set, load on the disk will increase as those pages have to be read back into memory to answer queries. We have found that this approach provides an accurate, low-overhead way of measuring the true RAM requirements of a workload.

2. *Combined Load Predictor:* We developed a model of CPU, RAM, and disk that allows Kairos to predict the combined resource requirements when multiple workloads are consolidated onto a single physical server. The many non-linearities of disk and RAM makes this task difficult. In particular, for disk I/O, we built a tool that creates a hardware-specific model of a given DBMS configuration, allowing us to predict how arbitrary OLTP/Web workloads will perform on that configuration. The accuracy of this model at predicting the combined disk requirements of multiple workloads is up to $30\times$ better than simply assuming that disk I/O combines additively. The reason is that two combined workloads perform many fewer I/Os than the sum of their individual I/Os: when combined, workloads share a single log, and can both benefit from group commit. Moreover, database systems perform a substantial amount of non-essential I/O during idle periods (e.g., flushing dirty pages to decrease recovery times)—in a combined workload, this activity can be curtailed without a substantial performance penalty.

3. *Consolidation Engine:* Finally, Kairos uses non-linear optimization techniques to place database partitions on back-end nodes to: (1) minimize the number of machines required to support a given workload mix, and (2) balance load across the back-end machines, while not exceeding machine capacities.

In addition to this placement functionality, another important feature of Relational Cloud is the capability to relocate database partitions across physical nodes. This relocation allows for scheduled maintenance and administration tasks, as well as to respond to load changes that entail the addition (or removal) of back-end machines. Relational Cloud aims to provide *live migration*, where data is moved between back-end nodes without causing downtime or adversely reducing performance. We are currently developing and testing a cache-like approach, where a new node becomes the new *master* for a portion of the data. Data is lazily fetched from the

old master as needed to support queries. In-flight transactions are redirected to the new master without being quiesced or killed.

5. PRIVACY

This section outlines *CryptDB*, the sub-system of Relational Cloud that guarantees the privacy of stored data by encrypting all tuples. The key challenge is executing SQL queries over the resulting encrypted data, and doing so efficiently. For example, a SQL query may ask for records from an employees table with a specific employee name; records whose salary field is greater than a given value; records joined with another table's rows, such as the employee's position; or even more complex queries, such as the average salary of employees whose position requires travel. Simply encrypting the entire database, or encrypting each record separately, will not allow the back-end DBMS to answer these kinds of queries. In addition, we would like a design that will allow DBAs (who operate Relational Cloud) to perform tuning tasks without having any visibility into the actual stored data.

Approach. The key idea in our approach is a notion we call *adjustable security*. We observe that there are many cryptographic techniques that we can build on to execute SQL queries, including randomized encryption (RND) and deterministic encryption (DET), as well as more recently developed order-preserving encryption (OPE) and homomorphic encryption (HOM). RND provides maximum privacy, such as indistinguishability under an adaptive chosen-plaintext attack without access to the key. However, RND does not allow any computation to be efficiently performed on the plaintext. DET provides a weaker privacy guarantee, because it allows a server to check plaintexts for equality by checking for equality of ciphertexts. OPE is even more relaxed in that it enables inequality checks and sorting operations, but has the nice property that the distribution of ciphertext is independent from the encrypted data values and also pseudorandom. Finally, HOM enables operations over encrypted data; in our case, we will mainly use additions and multiplications, which can be done efficiently.

Design. To implement adjustable security, our idea is to encrypt each value of each row independently into an *onion*: each value in the table is dressed in layers of increasingly stronger encryption, as shown in Figure 2. Each integer value is stored three times: twice encrypted as an onion to allow queries and once encrypted with homomorphic encryption for integers; each string type is stored once, encrypted in an onion that allows equalities and word searches and has an associated token allowing inequalities.

CryptDB starts the database out with all data encrypted with the most private scheme, RND. The JDBC client, shown in Figure 1, has access to the keys for all onion layers of every ciphertext stored on the server (by computing them based on a single master key). When the JDBC client driver receives SQL queries from the application, it computes the *onion keys* needed by the server to decrypt certain columns to the maximum privacy level that will allow the query execute on the server (such as DET for equality predicates). The security level dynamically adapts based on the queries that applications make to the server. We expect the database to converge to a certain security level when the application does not issue any more queries with new structures (only with different constants).

To simplify the management of onion keys, CryptDB encrypts all data items in a column using the same set of keys. Each layer of the onion has a different key (different from any other column), except for the lowest layer allowing joins (to allow meaningful comparisons of ciphertexts between two different columns as part of a join). The encryption algorithms are symmetric; in order for the server to remove a layer, the server must receive the symmetric

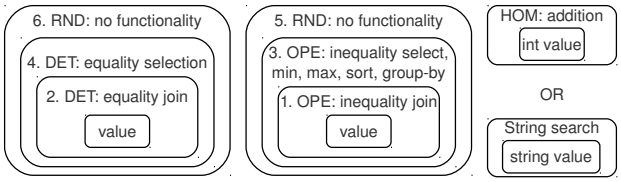


Figure 2: Onion layers of encryption.

onion key for that layer from the JDBC client. Once the server receives the key to decrypt an onion to a lower layer, it starts writing newly-decrypted values to disk, as different rows are accessed or queried. Once the entire column has been decrypted, the original onion ciphertext is discarded, since inner onion layers can support a superset of queries compared to outer layers.

For example, performing joins between tables requires the client to send keys to the server, which decrypts the joined columns to a layer that allows joins. There are two layers of DET and OPE encryption in the onion shown in Figure 2, corresponding to ciphertexts that can be used for comparisons for a single column (e.g., selection on equality to a given value with DET, selection based on comparison with OPE, etc.), and ciphertexts that can be used to join multiple columns together (i.e., using the same key). The database server then performs joins on the resulting ciphertexts as usual.

The key factor in the performance of CryptDB is ciphertext expansion. After the client issues a few queries, the server removes any unneeded onion layers of encryption, and from then on, it does not perform any more cryptographic operations. The server's only overhead is thus working with expanded tuples. If the ciphertext and plaintext are of equal length, most server operations, such as index lookups or scans, will take the same amount of time to compute. For DET, plaintext and ciphertext are equal in length, whereas for OPE, the ciphertext is double in length.

An example. To illustrate CryptDB's design, consider a TPC-C workload. Initially each column in the database is separately encrypted in several layers of encryption, with RND being the outer layer. Suppose the application issues the query `SELECT i_price, ... FROM item WHERE i_id=`N. The JDBC client will decrypt the `i_id` column to DET level 4 (Figure 2) by sending the appropriate decryption key to the server. Once that column is decrypted, the client will issue a SQL query with a `WHERE` clause that matches the DET-level `i_id` field to the DET-encrypted ciphertext of N. The query will return RND-encrypted ciphertexts to the JDBC client, which will decrypt them for the application. If the application's query requires order comparisons (e.g., looking for products with fewer than M items in stock), the JDBC client must similarly decrypt the onion to OPE level 3, and send an OPE ciphertext of the value M.

Suppose the application issues a join query, such as `SELECT c_discount, w_tax, ... FROM customer, warehouse WHERE w_id=c_w_id AND c_id=`N. To perform the join on the server, the JDBC client needs to decrypt the `w_id` and `c_w_id` columns to DET level 2 because the encryption should be deterministic not only within a column, but also across columns. The server can now perform the join on the resulting ciphertexts. Additionally, the JDBC client needs to decrypt `c_id` column to DET level 4, and send the DET-encrypted value N to the server for the other part of the `WHERE` clause.

Finally, CryptDB uses HOM encryption for server-side aggregates. For example, if a client asks `SELECT SUM(ol_amount) FROM order_line WHERE ol_o_id=`N, the server would need the keys to adjust the encryption of the `ol_amount` field to HOM, so that it can homomorphically sum up the encrypted `ol_amount` values, computing the total order amounts.

Table 1: Consolidation ratios for real-world datasets

Dataset	Input # Servers	Consolidated # Servers	Consolidation Ratio
TIG-CSAIL	25	2	12.5:1
Wikia	34	2	17:1
Wikipedia	40	6	6.6:1
Second Life	98	16	6.125:1

6. EXPERIMENTS

In this section, we describe several experiments we have run to evaluate Relational Cloud.

Consolidation/Multi-tenancy. We begin by investigating how much opportunity there is for consolidation in real-world database applications. We obtained the load statistics for about 200 servers from three data centers hosting the production database servers of Wikia.com, Wikipedia, and Second Life, and the load statistics from a cluster of machines providing shared services at MIT CSAIL.

Table 1 reports the consolidation levels we were able to achieve using the workload analysis and placement algorithms presented in Section 4. Here we used traces gathered over a 3-week duration, and found an allocation of databases to servers that Relational Cloud predicts will cause no server to experience more than 90% of peak load. The resulting ratios range from between 6:1 to 17:1 consolidation, demonstrating the significant reductions in capital and administrative expenses by adopting these techniques. One reason we are able to achieve such good consolidation is that our methods exploit the statistical independence and uncorrelated load spikes in the workloads.

In our second experiment, we compared the Relational Cloud approach of running multiple databases inside one DBMS to running each database in its own DBMS instance. Figure 3 (left) compares the performance of multiple databases consolidated inside a single DBMS that uses the entire machine, to the same server running one DBMS per database, in this case all in a single OS. We measure the maximum number of TPC-C instances than can run concurrently while providing a certain level of transaction throughput. Running a single DBMS allows 1.9–3.3× more database instances at a given throughput level. In Figure 3 (right) we compare the same single DBMS to multiple DBMSs, each running in a separate VM with its own OS. We measure the TPC-C throughput when there are 20 database instances on the physical machine—overall, a single DBMS instance achieves approximately 6× greater throughput for a uniform load and 12× when we skew the load (i.e., 50% of the requests are directed to one of the 20 databases).

The key reason for these results is that a single DBMS is much better at coordinating the access to resources than the OS or the VM hypervisor, enabling higher consolidation ratios on the same hardware. In particular, multiple databases in one DBMS share a single log and can more easily adjust their use of the shared buffer pool than in the multiple DBMS case where there are harder resource boundaries.

Scalability. We now measure how well the partitioning algorithm divides a database into independent partitions and how throughput scales as the database is spread across machines.

In this experiment we run TPC-C with a variable number of warehouses (from 16 to 128) and show what happens when database is partitioned and placed by Relational Cloud on 1 to 8 servers. The partitioner automatically splits the database by warehouse, placing 16 warehouses per server, and replicates the item table (which is never updated). We have manually verified that this partitioning is optimal. We measure the maximum sustained transaction throughput, which ranges from 131 transactions per second (TPS) with 16 warehouses on 1 machine up to 1007 TPS with 128 warehouses spread across 8 machines, representing a 7.7× speedup.

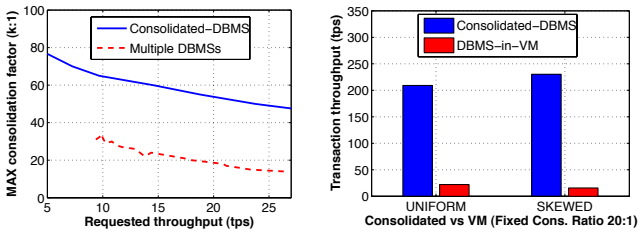


Figure 3: Multiplexing efficiency for TPC-C workloads

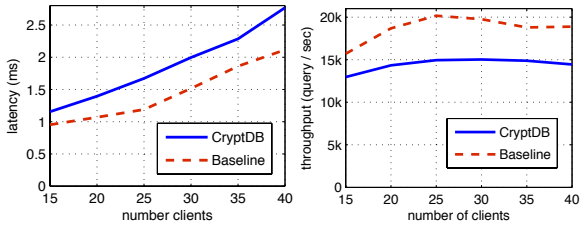


Figure 5: Impact of privacy on latency and throughput

We also measured the latency impact of our transaction coordinator on TPC-C, issuing queries to a single database with and without our system in place. On average, the Relational Cloud front-end adds 0.5 ms of additional latency per SQL statement (which for TPC-C adds up to 15 ms per transaction), resulting in a drop in throughput of about 12% from 149 TPS to 131 TPS.

The cost of privacy. CryptDB introduces additional latency on both the clients (for rewriting queries and encrypting and decrypting payloads), and on the server (due to the enlargement of values as a result of encryption.) We measured the time to process 100,000 statements (selects/updates) from a trace of TPC-C and recorded an average per statement overhead of 25.6 ms on the client side. We measure the effect that this additional latency is likely to cause in the next section. The server-side overhead is shown in Figure 5; the dashed line represents performance (latency–left, throughput–right) without CryptDB, and the solid line shows performance with CryptDB. Overall throughput drops by an average of 22.5%, which we believe will be an acceptable and tolerable performance degradation given the powerful privacy guarantees that are being provided.

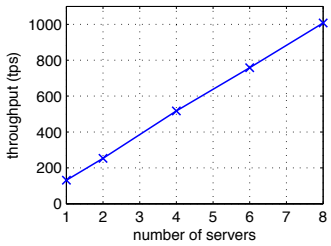


Figure 4: Scaling TPC-C.

The impact of latency.

In our final experiment, we measured the impact that additional latency between database clients and servers introduces on query throughput. This metric is relevant because in a Relational Cloud deployment, it may be valuable to run the database on a service provider like AWS (to provide a pool of machine for elastic scalability) with the application running in an internal data center across a wide-area network. Additionally, our privacy techniques depend on a (possibly remote) trusted node to perform query encoding and result decryption.

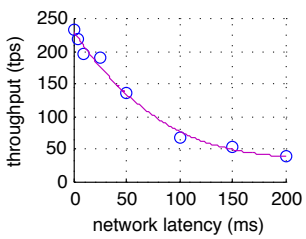


Figure 6: Impact of latency

We again ran TPC-C, this time with 128 warehouses and 512 client terminals (with no wait time). Figure 6 shows how aggregate throughput varies with increasing round-trip latency between the application and the DB server (we artificially introduced latency using a Linux kernel configuration parameter on the client machines.) We note that with latencies up to 20 ms, the drop in throughput is only about 12%, which is comparable to the latency of 10–20 ms we observe between AWS's east coast data center and MIT. The principal cause of this latency degradation is that locks are held for longer in the clients, increasing conflicts and decreasing throughput. Since TPC-C is a relatively high contention workload, it is likely that real-world workloads will experience lower throughput reductions.

Combining the results from the previous experiments, we expect an overall throughput reduction of about 40% when running CryptDB, our transaction coordinator, and a remote application, all at once. However, due to the linear scalability achieved via partitioning, we can compensate for this overhead using additional servers. As a result of the high consolidation ratios we measured on real applications, we still expect significant reduction in the overall hardware footprint, on the order of 3.5:1 to 10:1.

7. RELATED WORK

Scalable database services. Commercial cloud-based relational services like Amazon RDS and Microsoft SQL Azure have begun to appear, validating the market need. However, existing offerings are severely limited, supporting only limited consolidation (often simply based on VMs), lacking support for scalability beyond a single node, and doing nothing to provide any data privacy or ability to process queries over encrypted data. Some recent approaches [16, 17] try to leverage VM technologies, but our experiments show that these are significantly inferior in performance and consolidation to Relational Cloud's DBMS-aware DBaaS design.

Multi-tenancy. There have been several efforts to provide extreme levels of multi-tenancy [1, 12], aiming to consolidate tens of thousands of nearly inactive databases onto a single server, especially when those databases have identical or similar schemas. The key challenge of this prior work has been on overcoming DBMSs' limitations at dealing with extremely large numbers of tables and/or columns. These efforts are complementary to our work; we target heterogeneous workloads that do not share any data or schemas, and which impose significant load to the underlying DBMS in aggregate or even on their own—hence our focus on profiling and placement.

Scalability. Scalable database systems are a popular area for research and commercial activity. Approaches include NoSQL systems [3, 4, 2, 14], which sacrifice a fair amount of expressive power and/or consistency in favor of extreme scalability, and SQL-based systems that limit the type of transactions allowed [11, 7]. We differ from these in that we aim to preserve consistency and expressivity, achieving scalability via workload-aware partitioning. Our partitioning approach differs from prior work in that most prior work has focused on OLAP workloads and declustering [15, 19, 8].

Untrusted Storage and Computation. Theoretical work on homomorphic encryption [9] provides a solution to computing on encrypted data, but is too expensive to be used in practice. Systems solutions [10, 6, 18] have been proposed, but, relative to our solution, offer much weaker (at best) and compromised security guarantees, require significant client-side query processing and bandwidth consumption, lack core functionality (e.g. no joins), or require significant changes to the DBMS.

This related work is limited by space constraints; we plan to expand it in the camera ready should our paper be accepted.

8. CONCLUSION

We introduced Relational Cloud, a scalable relational database-as-a-service for cloud computing environments. Relational Cloud overcomes three significant challenges: *efficient multi-tenancy*, *elastic scalability*, and *database privacy*. For multi-tenancy, we developed a novel resource estimation and non-linear optimization-based consolidation technique. For scalability, we use a graph-based partitioning method to spread large databases across many machines. For privacy, we developed the notion of adjustable privacy and showed how using different levels of encryption layered as an "onion" can enable SQL queries to be processed over encrypted data. The key insight here is for the client to provide only the minimum decryption capabilities required by any given query. Based on our performance results, we believe that the Relational Cloud vision can be made a reality, and we look forward to demonstrating an integrated prototype at CIDR 2011.

9. REFERENCES

[1] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. In *SIGMOD*, 2008.

[2] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.

[3] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.

[5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. In *VLDB*, 2010.

[6] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing Confidentiality and Efficiency in Untrusted Relational DBMS. *CCS*, 2003.

[7] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An elastic transactional data store in the cloud. HotCloud, 2009.

[8] R. Freeman. *Oracle Database 11g New Features*. McGraw-Hill, Inc., New York, NY, USA, 2008.

[9] R. Gennaro, C. Gentry, and B. Parno. Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. *STOC*, 2010.

[10] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. *ACM SIGMOD*, 2002.

[11] P. Helland. Life beyond distributed transactions: An apostate's opinion. In *CIDR*, 2007.

[12] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *ICDE*, 2009.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1998.

[14] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1), 2009.

[15] D.-R. Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *ISJ*, 21, 1996.

[16] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1), 2010.

[17] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST*, 2009.

[18] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. Privacy-Preserving Computation and Verification of Aggregate Queries on Outsourced Databases. *HPL-2009-119*, 2009.

[19] D. C. Zilio. Physical database design decision algorithms and concurrent reorganization for parallel database systems. In *PhD thesis*, 1998.

Cloud Resource Orchestration: A Data-Centric Approach

Changbin Liu
University of Pennsylvania
3330 Walnut St, Philadelphia PA, USA
changbl@seas.upenn.edu

Yun Mao
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
maoy@research.att.com

Jacobus E. Van der Merwe
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
kobus@research.att.com

Mary F. Fernández
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
mff@research.att.com

ABSTRACT

Cloud computing provides users near instant access to seemingly unlimited resources, and provides service providers the opportunity to deploy complex information technology infrastructure, as a service, to their customers. Providers benefit from economies of scale and multiplexing gains afforded by sharing of resources through virtualization of the underlying physical infrastructure. However, the scale and highly dynamic nature of cloud platforms impose significant new challenges to cloud service providers. In particular, realizing sophisticated cloud services requires a cloud control framework that can orchestrate cloud resource provisioning, configuration, utilization and decommissioning across a distributed set of physical resources. In this paper we advocate a data-centric approach to cloud orchestration. Following this approach, cloud resources are modeled as structured data that can be queried by a declarative language, and updated with well-defined transactional semantics. We examine the feasibility, benefits and challenges of the approach, and present our design and prototype implementation of the Data-centric Management Framework (DMF) as a solution, with data models, query languages and semantics that are specifically designed for cloud resource orchestration.

1. INTRODUCTION

The efficiency and ubiquity of virtualization technologies on modern computer architecture have enabled widespread use of cloud computing. In particular, these Infrastructure-as-a-Service (IaaS) platforms provide cloud computing resources at the granularity of virtual machines (VMs) in both public cloud offerings, *e.g.*, Amazon EC2 [1], and enterprise-based private cloud instances. The latter is enabled by a variety of vendor offerings, or, indeed, an increasing array of open source cloud efforts.

While the basic IaaS model of providing VM instances on-demand, each with an operating system and set of applications of choice, remains the mainstay of cloud computing, more sophisticated cloud service abstractions are increasingly being offered and discussed in the community. Examples include combining cloud computing with virtual private network (VPN) technology to realize *virtual private cloud instances* [24], rapid cloning of VM instances to enable *cloud bursting* to deal with overload conditions [22, 18], *follow-the-sun* cloud services whereby VMs are migrated to be closer to where work is being performed [22] and using the ease of rapidly instantiating new VMs to provide cloud-based *disaster-recovery* services [23].

These more advanced cloud services share all the operational complexities associated with more basic cloud services, including resource allocation and placement, fault management, resource and service guarantees, image management, storage management etc. However, more sophisticated cloud services require the dynamic *orchestration* of resources to realize the service abstractions.

Cloud orchestration involves the creation, management and manipulation of cloud resources, *i.e.*, compute, storage and network, in order to realize user requests in a cloud environment, or to realize operational objectives of the cloud service provider. User requests are driven by the service abstraction and service logic that the cloud environment exposes to them. Examples of operational objectives that require orchestration functions include decreasing costs by consolidating physical resources and improving the ability to fulfill service level agreements (SLAs) by dynamically reallocating compute cycles and network bandwidth.

Cloud orchestration functions must be performed while dealing with service and operational concerns such as servicing large numbers of simultaneous user requests, enforcing policies that reflect service and engineering rules, and performing fault and error handling in a highly dynamic environment. Recognizing that similar problems are elegantly solved through well-known database methodologies and techniques, *i.e.* concurrency control, integrity constraint enforcement, and atomic transactions, we claim that an orchestration framework can and should incorporate database features: a formal, unified data model, a declarative query and constraint language, transactional semantics that provide atomicity, consistency, isolation and durability (ACID) properties, among others.

However, existing techniques for orchestration are rudimentary and meet few of the requirements listed above.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

First, control interfaces to resources range from simply editing textual or XML configuration files, to low-level command-line interfaces (CLIs), to procedure-oriented application programming interfaces (APIs). None provides a high-level data model for accessing or modifying resources in a consistent, system-wide manner. Currently, orchestration tasks are typically written as procedures in imperative programming or scripting languages and interact directly with resources via the myriad control interfaces. Consequently, specifying and enforcing of policies on a particular class of resources (e.g., physical compute servers) requires touching all the procedures that modify that resource class, replicating the policy throughout the code base. Even worse, exception handling, whether unexpected errors from resources or global constraint violations, is entirely ad hoc. For example, in a multi-step orchestration task, if a resource unexpectedly throws an error in some step, a typical implementation will fail-stop, leaving the system in an inconsistent state, or worse, silently ignore the error and execute subsequent steps, resulting in undefined behavior. Lastly, implementing deadlock-free concurrency control over distributed resources is challenging, especially in scripting languages, but is necessary to avoid race conditions between concurrent tasks that utilize shared resources.

The Data-centric Management Framework (DMF) is our answer to cloud orchestration. DMF is a cloud orchestration programming and execution framework, in which cloud operations can be easily specified and executed while ensuring that service and engineering constraints are satisfied in a system-wide manner. DMF models resources and their state as structured data, and further separates this data into logical and physical layers to avoid system misconfiguration and illegal operations. Orchestration procedures in DMF are transactional and provide well-defined semantics for accessing and updating resource data and for handling exceptions. In particular, DMF can atomically commit a group of operations, maintain consistency between the logical and physical layers, prevent misconfiguration and illegal resource manipulations by evaluating constraints before physical deployment, and provide race-free concurrent transactions. Realizing the benefits of a data-centric approach to cloud orchestration is, however, not a trivial task. In particular, database semantics need to be maintained while performing orchestration functions in a highly distributed environment, applying operations to resources that are not inherently atomic, and on a platform that is inherently much more volatile than conventional database systems. We have developed a prototype of DMF and performed initial experiments in our emulated wide-area cloud computing testbed, using Xen virtual machines [9], DRBD storage [21], and Juniper routers [2].

To illustrate the value of DMF's data-centric approach to orchestration we begin with an apparently simple cloud feature—VM migration across the wide area network—and show the complexity of its actual deployment. We then describe the unique challenges that a data-centric cloud orchestration approach poses outside the existing database research literature, and how they are addressed in the design of DMF. We conclude with a description of our prototype system, and a discussion of how DMF may potentially influence the future design of resource controllers.

2. BACKGROUND AND CHALLENGES

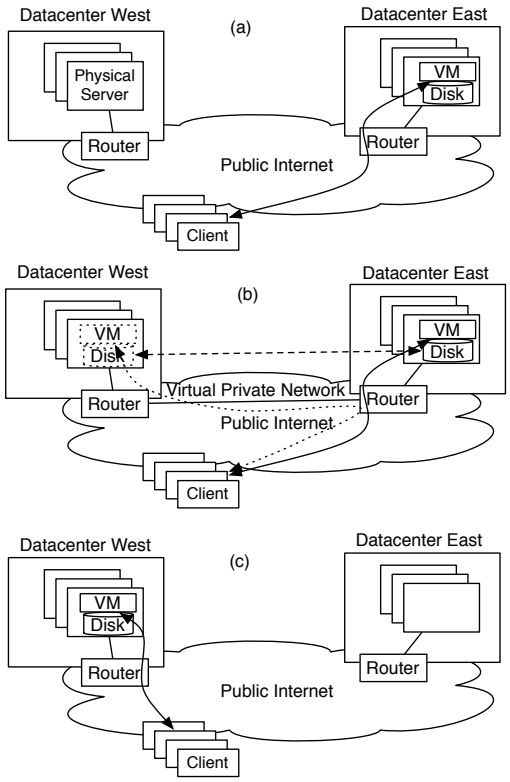


Figure 1: A multi data center, cross-domain cloud orchestration example

Figure 1 depicts our motivating example—a multi data center, cross-domain cloud orchestration scenario. The example illustrates the live migration of a VM from one cloud data center to another. This mechanism, for example, might be a part of the realization of a follow-the-sun service [22].

Figure 1 (a) shows the initial setup with a VM and its associated storage in Datacenter East and with clients accessing a service on the VM via the public Internet. The goal of our example is to "live migrate" the VM from Datacenter East to West to make it closer to the clients. Figures 1 (b) and (c) depict how this can be achieved. First a layer-2 VPN is established between the two data centers, and the storage associated with the VM is replicated to Datacenter West. [1] Then the VM itself is migrated from East to West.

During live VM migration, the IP address of the VM does not change, so existing application-level sessions are not disrupted [13]. As shown by the lower dotted line in Figure 1 (b), during migration, traffic between the clients and the VM follows a circuitous route via Datacenter East and the inter-datacenter VPN. Figure 1 (c) shows the final state after routing is updated so that traffic between the clients and the VM takes the direct path to Datacenter West.

Because routing in IP networks is asymmetric, there are two steps to updating network routing. First, traffic from

[1] In reality two separate VPNs might be used: A management VPN on which the storage and VM data is transferred between sites and a user VPN on which user traffic is carried.

the VM should use the router of its local data center for outgoing traffic, which can be readily achieved by changing the default route on the VM. Second, traffic from the clients to the VM needs to similarly follow the more direct path to Datacenter West. This can be achieved by advertising a more specific route to the migrated VM from Datacenter West.

The key point illustrated by this example is that realizing this dynamic service feature requires the orchestration of resources across different domains (compute, storage, and network), in a sequenced manner, and across geographically distributed data centers. Further, each of the operations described above is in fact relatively complex and thus susceptible to failure. Finally, a cloud orchestration platform would have to concurrently deal with potentially many similar orchestration requests.

Interestingly, similar problems in databases, such as making transactions atomic and controlling concurrent data manipulation have been solved through well-known methodologies and techniques. Solutions in the database literature, ranging from general semantics to specific algorithms, if not directly applicable, should at least inform solutions to related problems in cloud orchestration. To apply a data-centric approach to resource orchestration, however, several major differences from conventional data management must be addressed.

First, in resource orchestration, the main goal of updating data is to achieve a *side effect* of the state transition on a resource. For example, after setting a configuration of a network interface on a router to "enabled", one expects to be able to use the interface to send and receive traffic. Changing a replica of the configuration file outside of the router does not achieve the same result. The primary copy of the data on the device is the only copy that matters. In contrast, the purpose of writing data into a database is to be able to read it later. As a result, the data can be replicated to an arbitrary number of copies, on any media with any data format to serve the purpose. Therefore, data replication and caching must be used carefully in resource orchestration.

Second, an orchestration system manages data in the physical devices. Compared with data in the disk-based storage of databases, state in physical cloud resources is much more volatile. Given the scale, dynamics, and complexity of the cloud environment, crash or malfunction of a resource, such that its state changes, is sometimes inevitable. In addition, resource state may be tampered with, intentionally or even maliciously, via an out-of-band channel, circumventing the orchestration system.

Third, most state transitions in physical resources are expensive. For example, it takes from several seconds to minutes to commit a configuration on certain Juniper router models or perform a VM live migration, and not all state transitions are reversible. The orchestration system must take these factors into account when executing the orchestration procedures.

In the following sections, we will describe the design and implementation of DMF to address these challenges.

3. DATA-CENTRIC APPROACH

In this section we describe the design of Data-centric Management Framework (DMF) for cloud resource orchestration. Figure 2 depicts the architecture of DMF. In a nutshell,

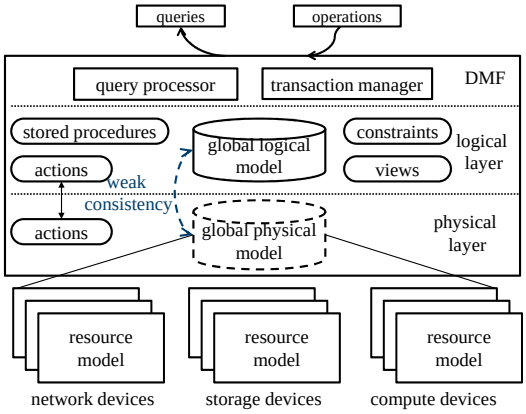


Figure 2: DMF architecture

DMF maintains a conceptually centralized data repository of all the resources being managed, which include compute, storage and network devices, as shown at the bottom of the figure. For every resource object, there are two copies that represent its state: the primary copy at the *physical layer* and the secondary copy at the *logical layer*. The primary copy is stored in the physical device such that read and write operations to the copy are translated into corresponding vendor-specific API calls. The secondary copy at the logical layer is an in-memory replica of the primary copy. DMF provides a weak, eventual data consistency between the two layers. Later in Section 3.3 we will explain why the separation of the two layers is necessary.

A user can specify *views* and integrity *constraints* in a declarative query language on top of the global data model. Views are used to reason about the current state in a system-wide fashion at a high level of abstraction. Constraints specify the policies that reflect service and engineering rules. Views and constraints can be materialized and are maintained by the *query processor*. *Actions* are the atomic operations that the resources provide, and are defined at both the physical and logical layers. A user can invoke transactions specified in *stored procedures* composed with queries, actions and other procedures to orchestrate cloud resources. They are executed by the *transaction manager* that enforces ACID properties.

We will now consider the DMF data model, language abstraction, transaction processing and consistency maintenance in turn.

3.1 Data Model

In DMF all resources are modeled as structured data. In this paper we only model non-volatile resource state, that is, state that changes as an effect of invoking orchestration operations on the devices, such as device configurations. Volatile state, like a server's CPU load or the latency between two routers, is read only and changes continuously. A data-centric approach to modeling volatile resource state has been studied in stream query processing systems [8, 14], and could be incorporated naturally into DMF, but is out of scope for this paper.

We adopt a hierarchical data model in which data is organized into a tree-like structure, as illustrated in Figure 4.

```
1  class LStorageRes(dmf.LogicalModel):
2      name = dmf.Attribute(str)
3      resRole = dmf.Attribute(StorageResRole)
4      @property # the primary key is attribute 'name'
5      def id(self): return self.name
6      @dmf.action
7      def setResRole(self, ctxt, resRole):
8          assert resRole in [StorageResRole.Primary,
9                             StorageResRole.Secondary]
10         origResRole = self.resRole
11         self.resRole = resRole
12         ctxt.appendlog(action="setResRole",
13                        args=[resRole],
14                        undo_action="setResRole",
15                        undo_args=[origResRole])
16 ...
17 class LStorageHost(dmf.LogicalModel):
18     hostname = dmf.Attribute(str)
19     resources = dmf.Many(LStorageRes)
20 ...
21 class LStorageRoot(dmf.LogicalModel):
22     hosts = dmf.Many(LStorageHost)
23     @dmf.view
24     def allResources(self):
25         return [(h.hostname, r.name, r.resRole)
26                 for h in self.hosts for r in h.resources]
27 ...
28 class LRoot(dmf.LogicalModel):
29     vmRoot = dmf.One(LVmRoot)
30     storageRoot = dmf.One(LStorageRoot)
31     @dmf.constraint
32     def vmAlwaysOnPrimary(self):
33         return [("VM not running on Primary Storage", vm)
34                 for host in self.vmRoot.hosts
35                 for vm in host.domains
36                 if self.storageRoot.hosts[host] \
37                     .resources[vm.storageRes].resRole
38                     != StorageResRole.Primary ]
39 ...
40 @dmf.proc
41 def migrate(root, vmName, srcHost, destHost):
42     resName = root.vmRoot.hosts[srcHost]\
43       .domains[vmName].storageRes
44     srcRes = root.storageRoot.hosts[srcHost]\
45       .resources[resName]
46     destRes = root.storageRoot.hosts[destHost]\
47       .resources[resName]
48     destRes.setResRole(StorageResRole.Primary)
49     root.vmRoot.migrate(vmName, srcHost, destHost)
50     srcRes.setResRole(StorageResRole.Secondary)
```

Figure 3: Sample code listing

Each tree node is an object representing an instance of an entity. An entity may have multiple *attributes* of primitive type, and multiple one-to-many and one-to-one *relations* to other entities, which occur as children nodes in the tree. An entity must have a primary key defined as a function of its attributes that uniquely identifies an object among its sibling objects in the tree.

The relational data model and SQL as the de facto standard in databases, are not entirely suitable for resource orchestration. For example, because resources are provided by multiple vendors, it is desirable to encapsulate state and provide modular interfaces in an object-oriented way. Heterogeneous data sources and limited APIs also favor the semi-structured or hierarchical data models, as exemplified in most integration middleware systems.

To illustrate the concepts of DMF, we use the example code in Figure 3, which is in a Python-style language. Although not complete, the code is very similar to the language implemented by DMF. The definitions of objects in the logical data model, in the classes `LRoot`, `LStorageRoot`, `LStorageHost`, `LStorageRes`, form the root node and storage-related branches. We will use this code sample throughout the remaining sections.

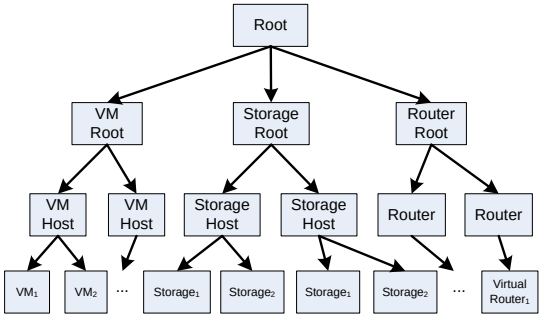


Figure 4: An example instance of the data model

3.2 Language

The programming language of DMF is a domain-specific language for query processing and data manipulation of structured data. It supports the following major constructs, *views, constraints, actions*, and *stored procedures*, which are elaborated below.

Orchestration tasks often require defining and querying views that, for example, inspect global network state, or that specify integrity constraints across multiple resources. Declarative query languages can express complex queries and constraints concisely and are highly amenable to optimization. Due to the tree-like data model, our query language syntax is similar to the FLWOR expressions in XQuery [7] with a subset of the XPath query capability. Because graph reachability queries are common in networked services, we also provide a transitive closure query operator and a more general fixpoint query operator that implements the semi-naïve evaluation algorithm [20]. A constraint in DMF is defined as a special type of view. It is satisfied if and only if it evaluates to an empty list. Otherwise, the list should contain information such as violation messages to help pinpoint the reasons behind the violations.

For example, on line 23 in Figure 3, the view `allResources` extracts attributes from nodes, returning a table of storage host names, and storage resource names and their roles. A more complicated constraint is defined on line 31, dictating that each VM must be running on a storage resource with primary role. [2]

For data manipulation, DMF provides the new concept of *action*, which models an *atomic* operation that is provided by a resource. Actions generalize the myriad APIs, ranging from file-based configurations, CLIs, to RPC-like APIs, provided by vendors to control the physical resources. Due to the varied semantics of these interfaces, DMF does not allow arbitrary insertions, deletions or updates to data, unless they are supported by the resources.

Due to the separation of the two layers, each action must be defined twice: one at the physical layer, which is transformed to the underlying API calls provided by the vendors, and the other at the logical layer, which describes the state

[2]A storage resource is usually a replicated data device that stores a VM image, and the storage resource associated with a running VM must have the resource role "primary".

transition in the data model. Preferably, an action is also decorated with a corresponding *undo* action. Undo actions are used to roll back transactions as described in Section 3.3. For example, on line 6 in Figure 3, the action `setResRole` is defined at the logical layer to change the resource role of a storage device. On lines 12–15, its corresponding undo action is written to the log: it is the same `setResRole` action, except with a different argument to set the resource role back to the original one.

3.3 Orchestration as Transactions

Transaction is the basic unit of orchestration in DMF. Transactions are atomic, consistent, isolated and durable and are realized in DMF as stored procedures. We describe how transactions are executed and how the separation of the physical and logical layers impacts ACID properties. We use the VM live migration procedure on lines 40–50 in Figure 3 as our example transaction. This corresponds to our previous cloud orchestration example in Section 2, with the difference that routing updating is omitted here for simplification.

A transaction is classified by its execution target layer as *logical-only*, *physical-only*, or *synchronized*, the latter meaning it is executed at both layers. Most orchestration tasks are synchronized transactions, because their purpose is both to satisfy constraints defined in the logical layer and to effect state change in the physical layer.

The execution of a synchronized transaction occurs in two phases. In the first phase, all operations in the transaction are executed at the logical layer, which include query evaluation and other actions. During the execution, an execution log is recorded.

In our example transaction, the queries on lines 42–47, access the relevant resource objects, and on lines 48–50, the 3-step orchestration (the destination storage resource role is set to primary, the VM is migrated to the destination, and the source storage resource role is set to secondary) is executed. The procedure is automatically enclosed in a transaction context (code omitted). Table 1 contains the execution log after the first phase.

At the end of the first phase, all integrity constraints are checked on the logical model. If any constraint is unsatisfied, the transaction is aborted, and the logical layer is rolled back to its state where the transaction began. This execution semantics guarantees that before a transaction begins and after it commits, the logical model is *internally consistent*, meaning that all integrity constraints are satisfied. The approach has the additional benefit that system misconfiguration and illegal operations are denied even before the physical resources are touched, thus avoiding the overhead of any unnecessary yet prohibitively expensive state transitions of physical resources.

If the first phase of a transaction succeeds, DMF executes the second phase at the physical layer. During this phase, since all state changes have already been handled in the logical model in previous phase, DMF simply re-plays all the actions in the execution log, executing the physical variant of each action. If all the physical actions succeed, the transaction returns as committed. This execution semantics guarantees that the transaction is *durable*, which means that the state of the physical resources have changed when the orchestration transaction is completed as committed.

If any action fails during the second phase, the transaction is aborted in both layers. At the logical layer, DMF rolls back to the original state, as it would if the first phase had aborted. At the physical layer, DMF selects all actions that have successfully executed, identifies the corresponding undo actions, and executes the undo actions in reverse chronological order. To achieve atomicity of transactions, each action in a transaction must have a corresponding undo action. If an action does not have an undo action, it can be executed stand-alone, but not within a transaction.

In our example, DMF executes the physical actions on the objects identified by their paths in the log. Suppose the first two actions succeed, but the third one fails. DMF executes the undo actions recorded in log, record #2 followed by record #1, to roll back the transaction. As a result, the VM is migrated back to the original location, and the destination storage resource role is reverted to secondary.

Once all undo actions complete, the transaction is terminated as aborted. If an error occurs during the undo phase, the transaction is terminated as failed. In this case, the logical layer is rolled back to an internally consistent state, however, there may be inconsistency *between* the physical and logical layers.

We note that in this execution model, all-or-nothing atomicity is always guaranteed at the logical layer. At the physical layer, it is enforceable if (1) each physical action is atomic, (2) each physical action is reversible with an undo action, (3) all undo actions succeed during rollback, (4) the resources are not volatile during transaction execution.

The first two assumptions can be largely satisfied at design time. According to our experience, most actions, such as resource allocation and configuration are reversible. [3] For (3), because an action and its undo action are symmetric, the undo action usually has a high probability of success given the fact that its action has been successfully executed in the recent past during the transaction.

3.4 Cross-layer Consistency Maintenance

DMF provides a weak, eventual consistency model for cross-layer consistency maintenance. A synchronized transaction maintains cross-layer consistency if the data is cross-layer consistent before the transaction starts, and the transaction can be atomically committed or aborted, under the assumptions described above.

However, in addition to failed undo actions, out-of-band changes to physical devices may cause cross-layer inconsistencies. For example, an operator may add or decommission a physical resource without making the change visible to DMF. Or an operator may log in to a device directly and change its state via the CLI. Furthermore, because resources are complex physical devices, a crash or system malfunction may change the resource's physical state without DMF's knowledge. Whatever the causes, these out-of-band changes occur in practice, and DMF must be able to gracefully handle inconsistencies.

Specifically, in DMF, inconsistency can be automatically identified when a physical undo action fails in a transaction, or can be detected by periodically comparing the data between the two layers. Once an inconsistency is detected on a node in the tree, no more synchronized transactions are allowed at that node or its children until the inconsistency

[3] Although not all physical actions can be undone. E.g., after a server reboots, there is no (easy) way to return the server to its pre-reboot state.

log record #	resource object path	action	args	undo action	undo args
1	/storageRoot/dest/vmRes	setResRole	[primary]	setResRole	[secondary]
2	/vmRoot	migrate	[vmName,src,dest]	migrate	[vmName,dest,src]
3	/storageRoot/src/vmRes	setResRole	[secondary]	setResRole	[primary]

Table 1: An example of execution log for VM migration

is reconciled.

To reconcile inconsistencies, logical-only and physical-only transactions are applied in a disciplined and reliable way. A DMF user can invoke a logical-only transaction to "reload" the logical model from the physical state, or invoke a physical-only transaction to "repair" the resources by aligning them with the logical model. Before a logical-only transaction commits, DMF checks all integrity constraints. If any constraints are violated, DMF aborts the transaction. To execute a physical-only transaction, at the beginning DMF first "reloads" the physical state into the logical layer, then executes the rest of the transaction as if it were a synchronized transaction.

DMF itself does not require a specific consistency maintenance schedule, leaving that schedule to the user. One can periodically invoke repair procedures, or in the case of a new device addition, for example, manually invoke a reload procedure to add that device to the system.

3.5 Summary

To summarize, DMF provides a data-centric programming and execution framework for transactional cloud resource orchestration. DMF provides ACID properties that closely resemble those of SQL databases at the logical layer with a strong consistency guarantee, and "best-effort" transactional semantics at the physical layer to cope with the limitations of the physical devices. A weak, eventual consistency model is used to reconcile the cross-layer differences.

4. PROTOTYPE

4.1 Implementation

We have implemented a prototype of DMF in Python. The primary goal of the prototype is to validate our hypothesis and explore the benefits of the data-centric approach.

The prototype system runs on a centralized server. The resource models, views, constraints, actions and stored procedures are all specified in corresponding Python programs following the code style in Figure 3. Most of the queries are expressed in the form of declarative list comprehensions and string-based path queries, all embedded in Python. The results of a view or a constraint can be optionally materialized to speed up subsequent queries.

We extensively use Python's meta programming and decorator techniques to hide implementation details, like how the model instances are stored, queried, and updated, how the logical and physical layers are separated, and how to execute, commit and abort transactions.

An obvious alternative to designing a new language would be to use an existing query language, like XQuery, which natively supports a tree-structured data model and a compact syntax for querying XML. XQuery's syntax for updates is cumbersome, however, and its XML-friendly syntax is not easily extended to support the DMF constructs described above. In addition, we wanted our network engineers, the first DMF developers, to be able to learn DMF easily. Python and its rich libraries are a familiar and expedient choice. In the end, DMF provides many of the benefits of XQuery, without burdening users with having to learn a new language syntax and library.

We fully implemented the transaction execution model described in Section 3.3. All transactions are serialized internally to provide isolation. A more sophisticated concurrency control model is our future work.

Specifically, we have implemented models for DRBD storage [21], Xen virtual machine hypervisor [9], and Juniper routers [2] in DMF. The three classes of resources provide very different APIs for orchestration. DRBD relies on the text based configuration files and a command-line interface to update resource roles and other state in the kernel. Xen provides its own APIs, but is also compatible with a more generic set of virtualization APIs provided by the `libvirt` library [3] that works with a variety of virtualization technologies, including Xen, VMWare, KVM, etc. Juniper routers use the NETCONF protocol [4] for router configuration. The configuration format is in XML with a predefined XML schema. Therefore, the process of building data models for DRBD and `libvirt` on Xen are entirely manual, such as designing entities and relationships, and wrapping their API calls to actions in DMF. In contrast, because Juniper already provides the XML schema, we are able to automatically generate models from the schema into the DMF modeling language (3197 models imported in total). We still have to manually write actions for operations like configuration commit, and constraints such as network protocol dependencies.

DMF exports an XML-RPC interfaces so that cloud orchestration applications can invoke stored procedures to realize cloud functions. We also have built an interactive command-line shell for rapid testing, debugging and demonstration purposes.

4.2 Preliminary Evaluation

To evaluate the performance of DMF, we experiment with a representative transaction—live virtual machine (VM) migration across a wide area network (WAN) as described in Section 2 and depicted in Figure 1, and measure the transaction execution time for both logical and physical layers. As mentioned above, this transaction consists of several actions, which include destination and source storage resource role setting, live VM migrating, and route updating.

We emulate the cloud environment that DMF controls and orchestrates on ShadowNet, our operational wide-area testbed [12]. In particular we create a slice in ShadowNet for DMF that consists of a server and a router in each of two geographically distributed ShadowNet locations, *i.e.* in Illinois (IL) and California (CA). The router in each location provides access to the public Internet and is used to create an inter-data center VPN for this slice. The physical servers are configured so that DRBD storage replication can be per-

data model	average	stdev
logical layer	0.069	0.026
physical layer	38.927	1.934

Table 2: Average and standard deviation of the transaction execution time (in seconds) for VM live migration.

formed between them and the to-be-migrated VM runs on this storage. The VM is allocated with 512MB memory in size, which is the dominant factor affecting the actual migration time.

To make the result more accurate, we in total migrate the VM 10 times and compute the average transaction execution time, as well as the standard deviation. Table 2 lists the experimental results. We note that, as expected, most of the transaction time is spent in physical layer, where actual management operations are performed on the physical devices, while the time for logical layer is negligible, demonstrating the efficiency of DMF implementation.

5. RELATED WORK

Database technologies are routinely used as part of system management and operations. One notable class of existing work, *e.g.* NetDB [10] in network configuration management, uses a relational database to store device configuration snapshots, where one can write queries to audit and perform static analysis of existing configurations in an offline fashion. However, NetDB is a data warehouse, not designed for network resource orchestration. Transaction processing [15] as another database technology also has received more attentions recently as a programming paradigm in system areas. Since Microsoft Windows Vista, the Kernel Transaction Manager (KTM) [6] enables the development of applications that use transactions, for instance, to implement transactional file system and transactional registry. When failure happens, transactions are rolled back to restore system state. Transactional OS (TxOS) [19] explores adding transactions to the OS system calls. A system transaction executes a series of system calls in isolation and atomically publishes the effects to the rest of the OS.

There are several related frameworks proposed for management and orchestration for large-scale systems. Autopilot [17] is a data center software management infrastructure from Microsoft for automating software provisioning, monitoring and deployment. Similar to DMF, it has repair actions to deal with faulty software and hardware. Its periodic repair procedures maintain weak consistency between the provisioning data repository and the deployed software code. From the open-source community, Puppet [5] is a data center automation and configuration management framework using a custom and user-friendly declarative language for server configuration specification. In contrast to DMF, other than having different scopes, Autopilot does not provide a transactional programming interface. Puppet has a transactional layer, but not in the sense of enforcing ACID properties. Instead it allows user to visually examine the detailed operations before a transaction is submitted as a dry run. Once executed, a transaction is not guaranteed to be atomically committed.

Finally, in our earlier work, COOLAID [11] proposes a similar vision of data-centric network configuration management. COOLAID manages router configurations and adopts the relational data model and Datalog-style query language. In contrast, DMF has the significantly expanded scope of cloud resource orchestration where a diverse set of devices are managed. Facing several new challenges, DMF not only utilizes a different data model and query language to improve usability, but also re-architect the system to provide well-defined transaction executions on top of the separated logical and physical layers, in refined transactional semantics.

6. CONCLUSIONS

We presented DMF as the first attempt in adopting a data-centric approach that combines the uses of structured data models, a declarative query language and transactional ACID semantics to support cloud resource orchestration. We argue that these methodologies, mature and well-understood in databases, address many challenges imposed by emerging cloud computing services. Specifically, DMF offers transactional orchestration to atomically commit a group of orchestration tasks and provides well-defined semantics for unexpected error handling. Separating the resource data model into logical and physical layers, DMF maintains cross-layer consistency between the two, and further prevents the system from misbehaving by enforcing integrity constraints as policies. Ultimately only operational experience will tell how successful our approach is. However, we expect that, compared to conventional systems built in imperative languages, DMF's uniform declarative language for query and constraint specification will result in better code scalability and maintainability as the number and heterogeneity of resources increases. We have implemented a preliminary prototype of DMF and evaluated it within an emulated wide-area cloud environment that involves compute, storage, and network devices. The prototype demonstrates the feasibility of DMF design and its practical aspects.

Our future work includes: (1) exploring concurrency control algorithms to improve parallelism under simultaneous transactions from multiple clients; (2) decentralizing DMF architecture to realize high system scalability, reliability and availability; (3) deploying and evaluating DMF in geographically distributed large-scale data centers; (4) adopting query optimization techniques and incremental view maintenance [16] to improve performance; (5) and building more sophisticated cloud services in the framework to further validate our hypothesis and explore new opportunities.

7. APPENDIX: DEMONSTRATION

At the conference we will demonstrate live virtual machine (VM) migration across a wide area network (WAN) as described in Section 2, depicted in Figure 1 and evaluated in Section 4.2.

To demonstrate DMF's capability of orchestrating live VM migration, we run a game server in the VM which starts off in the Illinois (IL) data center. All game clients connects to this server via the IL router. For the demonstration purpose we assume that as more gaming clients connect to the server it becomes apparent that the data center in California (CA) would be more optimally positioned to serve them. As such we migrate the game server from IL to CA. This migration simulates a type of follow-the-sun cloud service in which the server migrates to the time zone where the major-

ity of clients are located, to reduce latency and improve their gaming experience. We run two processes to emulate game clients connecting respectively from IL and CA. A third process runs as DMF administrator and executes the migration transaction.

For the demonstration, the DMF control interface is a simple visualization tool, which displays the cloud resources and their state. The orchestration process proceeds as a transactional execution of (i) setting up the inter-data center VPN, (ii) establishing storage replication between the two data centers, (iii) performing VM migration, (iv) updating routing so that traffic to/from game clients follow the most direct path to the gaming server. As the migration transaction progresses, the visualization is updated to illustrate state change.

To demonstrate DMF transactional semantics, we emulate the failure of route re-configuration. Because route changing is the last step of the migration transaction, all previous steps are rolled back. The rollback operations are reported by the administration process.

If time permits, we will demonstrate another feature of DMF: consistency maintenance between logical and physical layers (described in Section 3.4) after the host machines have crashed and reset their state.

If there is a problem with the Internet connection at the venue, we will play a pre-recorded video for the demo.

Acknowledgments

The authors would like to thank Xu Chen, Carsten Lund, Boon Thau Loo, Emmanuil Mavrogiorgis, Edwin Lim, and the anonymous reviewers for their helpful comments on the earlier versions of the paper. This work is supported in part by the AT&T Labs Research summer student internship program and NSF grant CCF-0820208.

8. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Juniper Networks. `www.juniper.net`.

[3] Libvirt: The virtualization API. `http://libvirt.org`.

[4] Network configuration (netconf). `http://www.ietf.org/html.charters/netconf-charter.html`.

[5] Puppet: A Data Center Automation Solution. `http://www.puppetlabs.com/`.

[6] Windows Kernel Transaction Manager. `http://msdn.microsoft.com/en-us/library/bb986748(VS.85).aspx`.

[7] XQuery: the W3C XML Query Language. `http://www.w3.org/TR/xquery`.

[8] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan 2005.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[10] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting EDGE of IP router configuration. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, November 2003.

[11] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proceedings of the 6th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2010.

[12] X. Chen, Z. M. Mao, and J. Van der Merwe. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[14] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[16] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993.

[17] M. Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[18] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[19] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[20] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2002.

[21] P. Reisner. DRBD - Distributed Replication Block Device. In *9th International Linux System Technology Conference*, September 2002.

[22] J. Van der Merwe, K. Ramakrishnan, M. Fairchild, A. Flavel, J. Houle, H. A. Lagar-Cavilla, and J. Mulligan. Towards a ubiquitous cloud computing infrastructure. In *Proceedings of the IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*, May 2010.

[23] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2010.

[24] T. Wood, A. Gerber, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. The case for enterprise-ready virtual private clouds. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2009.

Consistency Analysis in Bloom: a CALM and Collected Approach

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak

{palvaro, nrc, hellerstein, wrm}@cs.berkeley.edu
University of California, Berkeley

ABSTRACT

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications.

We address this situation with the *CALM* principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce *Bloom*, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a program analysis technique that identifies *points of order* in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

1. INTRODUCTION

Until fairly recently, distributed programming was the domain of a small group of experts. But recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [6, 12, 13, 18, 24]). This approach enables machines to continue operating in the face of temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

Merging the best of these traditions, it would be ideal to have a robust theory and practical tools to help programmers reason about and manage high-level program properties in the face of loosely coordinated consistency. In this paper we demonstrate significant progress in this direction. Our approach is based on the use of a declarative language and program analysis techniques that enable both static analyses and runtime annotations of consistency. We begin by introducing the *CALM* principle, which connects the theory of non-monotonic logic to the need for distributed coordination to achieve consistency. We present an initial version of our *Bloom* declarative language, and translate concepts of monotonicity into a practical program analysis technique that detects potential consistency anomalies in distributed Bloom programs. We then show how such anomalies can be handled by a programmer during the development process, either by introducing coordination mechanisms to ensure consistency or by applying program rewrites that can track inconsistency "taint" as it propagates through code. To illustrate the Bloom language and the utility of our analysis, we present two case studies: a replicated key-value store and a fault-tolerant shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present the connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

A key problem in distributed programming is reasoning about the consistent behavior of a program in the face of *temporal nondeterminism*: the delay and re-ordering of messages and data across nodes. Because delays can be unbounded, analysis typically focuses on "eventual consistency" after all messages have been delivered [26]. A sufficient condition for eventual consistency is *order independence*:

5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9–12, 2011, Asilomar, California, USA.

the independence of program execution from temporal nondeterminism.

Order independence is a key attribute of declarative languages based on sets, which has led most notably to the success of parallel databases and web search infrastructure. But even set-oriented languages can require a degree of ordering in their execution if they are sufficiently expressive. The theory of relational databases and logic programming provides a framework to reason about these issues. *Monotonic* programs—e.g., programs expressible via selection, projection and join (even with recursion)—can be implemented by streaming algorithms that incrementally produce output elements as they receive input elements. The final order or contents of the input will never cause any earlier output to be "revoked" once it has been generated.[1] *Non-monotonic* programs—e.g., those that contain aggregation or negation operations—can only be implemented correctly via blocking algorithms that do not produce any output until they have received all tuples in logical partitions of an input set. For example, aggregation queries need to receive entire "groups" before producing aggregates, which in general requires receiving the entire input set.

The implications for distributed programming are clear. Monotonic programs are easy to distribute: they can be implemented via streaming set-based algorithms that produce actionable outputs to consumers while tolerating message reordering and delay from producers. By contrast, even simple non-monotonic tasks like counting are difficult in distributed systems. As a mnemonic, we say that *counting requires waiting* in a distributed system: in general, a complete count of distributed data must wait for all its inputs, including stragglers, before producing the correct output.

"Waiting" is specified in a program via *coordination logic*: code that (a) computes and transmits auxiliary information from producers to enable the recipient to determine when a set has completely arrived across the network, and (b) postpones production of results for consumers until after that determination is made. Typical coordination mechanisms include sequence numbers, counters, and consensus protocols like Paxos or Two-Phase Commit.

Interestingly, these coordination mechanisms themselves typically involve counting. For example, Paxos requires counting messages to establish that a majority of the members have agreed to a proposal; Two-Phase Commit requires counting to establish that all members have agreed. Hence we also say that *waiting requires counting*, the converse of our earlier mnemonic.

Our observations about waiting and counting illustrate the crux of what we call the *CALM* principle: the tight relationship between Consistency And Logical Monotonicity. Monotonic programs *guarantee* eventual consistency under any interleaving of delivery and computation. By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that "wait" until inputs can be guaranteed to be complete.

We typically wish to minimize the use of coordination, because of well-known concerns about latency and availability in the face of message delays and network partitions. We can use the CALM principle to develop checks for distributed consistency in logic languages, where conservative tests for monotonicity are well understood. A simple syntactic check is often sufficient: if the program does not contain any of the symbols in the language that correspond to non-monotonic operators (e.g., `NOT IN` or aggregate symbols), then it is monotonic and can be implemented without coordination, regardless of any read-write dataflow dependencies in the code. As students of the logic programming literature will recognize [19, 20, 21], these conservative checks can be refined further to consider semantics of predicates in the language. For example, the expression "`MIN(x)` < 100" is monotonic despite containing an aggregate, by virtue of the semantics of `MIN` and $<$: once a subset S satisfies this test, any superset of S will also satisfy it. Many refinements along these lines exist, increasing the ability of program analyses to verify monotonicity.

In cases where an analysis cannot guarantee monotonicity of a whole program, it can instead provide a conservative assessment of the points in the program where coordination may be required to ensure consistency. For example, a shallow syntactic analysis could flag all non-monotonic predicates in a program (e.g., `NOT IN` tests or predicates with aggregate values as input). The loci produced by a non-monotonicity analysis are the program's *points of order*. A program with non-monotonicity can be made consistent by including coordination logic at its points of order.

The reader may observe that because "waiting requires counting," adding a code module with coordination logic actually increases the number of syntactic points of order in a program. To avoid this problem, the coordination module itself must be verified for order independence, either manually or via a refined monotonicity test during analysis. When the verification is done by hand, annotations can inform the analysis tool to skip the module in its analysis, and hence avoid attempts to coordinate the coordination logic.

Because analyses based on the CALM principle operate with information about program semantics, they can avoid coordination logic in cases where traditional read/write analysis would require it. Perhaps more importantly, as we will see in our discussion of shopping carts (Section 5), logic languages and the analysis of points of order can help programmers redesign code to reduce coordination requirements.

3. BUD: BLOOM UNDER DEVELOPMENT

Bloom is based on the conjecture that many of the fundamental problems with parallel programming come from a legacy of ordering assumptions implicit in classical von Neumann architectures. In the von Neumann model, state is captured in an ordered array of addresses, and computation is expressed via an ordered list of instructions. Traditional imperative programming grew out of these pervasive assumptions about order. Therefore, it is no surprise that popular imperative languages are a bad match to parallel and distributed platforms, which make few guarantees about order of execution and communication. By contrast, set-oriented approaches like SQL and batch dataflow approaches like MapReduce translate better to architectures with loose control over ordering.

Bloom is designed in the tradition of programming styles that are "disorderly" by nature. State is captured in unordered sets. Computation is expressed in logic: an unordered set of declarative rules, each consisting of an unordered conjunction of predicates. As we discuss below, mechanisms for imposing order are available when needed, but the programmer is provided with tools to evaluate the need for these mechanisms as special-case behaviors, rather than a default model. The result is code that runs naturally on distributed machines with a minimum of coordination overhead.

Unlike earlier efforts such as Prolog, active database languages, and our own Overlog language for distributed systems [16], Bloom is *purely declarative*: the syntax of a program contains the full specification of its semantics, and there is no need for the programmer to understand or reason about the behavior of the evaluation engine. Bloom is based on a formal temporal logic called Dedalus [3].

The prototype version of Bloom we describe here is embodied in an implementation we call *Bud* (Bloom Under Development). Bud

[1] Formally, in a monotonic logic program, any true statement continues to be true as new axioms—including new facts—are added to the program.

Type	Behavior
table	A collection whose contents persist across timesteps.
scratch	A collection whose contents persist for only one timestep.
channel	A scratch collection with one attribute designated as the *location specifier*. Tuples "appear" at the network address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The definition of a periodic collection is parameterized by a `period` in seconds; the runtime system arranges (in a best-effort manner) for tuples to "appear" in this collection approximately every `period` seconds, with a unique `id` and the current wall-clock time.
interface	A scratch collection specially designated as an interface point between modules.

Op	Valid lhs types	Meaning
=	**scratch**	rhs defines the contents of the lhs for the current timestep. lhs must not appear in lhs of any other statement.
<=	**table**, **scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table**, **scratch**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some non-deterministic future time.

Figure 1: Bloom collection types and operators.

is a domain-specific subset of the popular Ruby scripting language and is evaluated by a stock Ruby interpreter via a `Bud` Ruby class. Compared to other logic languages, we feel it has a familiar and programmer-friendly flavor, and we believe that its learning curve will be relatively flat for programmers familiar with modern scripting languages. Bud uses a Ruby-flavored syntax, but this is not fundamental; we have experimented with analogous Bloom embeddings in other languages including Python, Erlang and Scala, and they look similar in structure.

3.1 Bloom Basics

Bloom programs are bundles of declarative statements about collections of "facts" or tuples, similar to SQL views or Datalog rules. A statement can only reference data that is local to a node. Bloom statements are defined with respect to atomic "timesteps," which can be implemented via successive rounds of evaluation. In each timestep, certain "ground facts" exist in collections due to persistence or the arrival of messages from outside agents (e.g., the network or system clock). The statements in a Bloom program specify the derivation of additional facts, which can be declared to exist either in the current timestep, at the very next timestep, or at some non-deterministic time in the future at a remote node.

A Bloom program also specifies the way that facts persist (or do not persist) across consecutive timesteps on a single node. Bloom is a side-effect free language with no "mutable state": if a fact is defined at a given timestep, its existence at that timestep cannot be refuted by any expression in the language. This technicality is key to avoiding many of the complexities involved in reasoning about earlier "stateful" rule languages. The paper on Dedalus discusses these points in more detail [3].

3.2 State in Bloom

Bloom programs manage state using five collection types described in the top of Figure 1. A collection is defined with a relational-style schema of named columns, including an optional subset of those columns that forms a primary key. Line 15 in Figure 2 defines a collection named `send_buf` with four columns `dst`, `src`, `ident`, and `payload`; the primary key is `(dst, src, ident)`. The type system for columns is taken from Ruby, so it is possible to have a column based on any Ruby class the programmer cares to define or import (including nested Bud collections). In Bud, a tuple in a collection is simply a Ruby array containing as many elements as the columns of the collection's schema. As in other object-relational ADT schemes like Postgres [23], column values can be manipulated using their own (non-destructive) methods. Bloom also provides for nesting and unnesting of collections using standard Ruby constructs like `reduce` and `flat_map`. Note that collections in Bloom provide set semantics—collections do not contain duplicates.

```
0  module DeliveryProtocol
1    def state
2      interface input, :pipe_in,
3        ['dst', 'src', 'ident'], ['payload']
4      interface output, :pipe_sent,
5        ['dst', 'src', 'ident'], ['payload']
6    end
7  end

9  module ReliableDelivery
10   include DeliveryProtocol

12   def state
13     channel :data_chan, ['@dst', 'src', 'ident'], ['payload']
14     channel :ack_chan, ['@src', 'dst', 'ident']
15     table :send_buf, ['dst', 'src', 'ident'], ['payload']
16     periodic :timer, 10
17   end

19   declare
20   def send_packet
21     send_buf <= pipe_in
22     data_chan <~ pipe_in
23   end

25   declare
26   def timer_retry
27     data_chan <~ join([send_buf, timer]).map{|p, t| p}
28   end

30   declare
31   def send_ack
32     ack_chan <~ data_chan.map{|p| [p.src, p.dst, p.ident]}
33   end

35   declare
36   def recv_ack
37     got_ack = join [ack_chan, send_buf],
38                    [ack_chan.ident, send_buf.ident]
39     pipe_sent <= got_ack.map{|a, sb| sb}
40     send_buf <- got_ack.map{|a, sb| sb}
41   end
42 end
```

Figure 2: Reliable unicast messaging in Bloom.

The persistence of a tuple is determined by the type of the collection that contains the tuple. **scratch** collections are useful for transient data like intermediate results and "macro" definitions that enable code reuse. The contents of a **table** persist across consecutive timesteps (until that persistence is interrupted via a Bloom statement containing the `<-` operator described below). Although there are precise declarative semantics for this persistence [3], it is convenient to think operationally as follows: scratch collections are "emptied" before each timestep begins, tables are "stored" collections (similar to tables in SQL), and the `<-` operator represents batch deletion before the beginning of the next timestep.

The facts of the "real world," including network messages and the passage of wall-clock time, are captured via **channel** and **periodic** collections; these are scratch collections whose contents "appear" at non-deterministic timesteps. The paper on Dedalus delves deeper

Method	Description
`bc.map`	Takes a code block and returns the collection formed by applying the code block to each element of `bc`.
`bc.flat_map`	Equivalent to `map`, except that any nested collections in the result are flattened.
`bc.reduce`	Takes a `memo` variable and code block, and applies the block to `memo` and each element of `bc` in turn.
`bc.empty?`	Returns true if `bc` is empty.
`bc.include?`	Takes an object and returns true if that object is equal to any element of `bc`.
`bc.group`	Takes a list of grouping columns, a list of aggregate expressions and a code block. For each group, computes the aggregates and then applies the code block to the group/aggregation result.
`join`, `leftjoin`, `outerjoin`, `natjoin`	Methods of the Bud class to compute join variants over BudCollections. `join`, `leftjoin` and `outerjoin` take an array of collections to join, as well as a variable-length list of arrays of join conditions. The natural join `natjoin` takes only the array of BudCollection objects as an argument.

Figure 3: Commonly used methods of the BudCollection class.

into the logical semantics of this non-determinism [3]. Note that failure of nodes or communication is captured here: it can be thought of as the repeated "non-appearance" of a fact at every timestep. Again, it is convenient to think operationally as follows: the facts in a channel are sent to a remote node via an unreliable transport protocol like UDP; the address of the remote node is indicated by a distinguished column in the channel called the *location specifier* (denoted by the symbol `@`). The definition of a periodic collection instructs the runtime to "inject" facts at regular wall-clock intervals to "drive" further derivations. Lines 13 and 16 in Figure 2 contain examples of channel and periodic definitions, respectively.

The final type of collection is an **interface**, which specifies a connection point between Bloom modules. Interfaces are described in Section 3.4.

3.3 Bloom Statements

Bloom statements are declarative relational expressions that define the contents of derived collections. They can be viewed operationally as specifying the insertion or accumulation of expression results into collections. The syntax is:

<collection-variable> <op> <collection-expression>

The bottom of Figure 1 describes the five operators that can be used to define the contents of the left-hand side (lhs) in terms of the right-hand side (rhs). As in Datalog, the lhs of a statement may be referenced recursively in its rhs, or recursion can be defined mutually across statements.

In the Bud prototype, both the lhs and rhs are instances of (a descendant of) a Ruby class called `BudCollection`, which supports several useful methods for manipulating collections (Figure 3).[2] The rhs of a statement typically invokes `BudCollection` methods on one or more collection objects to produce a derived collection. The most commonly used method is `map`, which applies a scalar operation to every tuple in a collection; this can be used to implement relational selection and projection. For example, line 32 of Figure 2 projects the `data_chan` collection to its `src`, `dst`, and `ident` fields. Multiway joins are specified using the `join` method, which takes a list of input collections and an optional list of join conditions. Lines 37–38 of Figure 2 show a join between `ack_chan` and `send_buf`. Syntax sugar for natural joins and outer joins is also provided. `BudCollection` also defines a `group` method similar to SQL's `GROUP BY`, supporting the standard SQL aggregates; for example, lines 15–17 of Figure 14 compute the count of unique `reqid` values for every combination of values for `session`, `item` and `action`.

Bloom statements are specified within method definitions that are flagged with the `declare` keyword (e.g., line 20 of Figure 2). The semantics of a Bloom program are defined by the union of its `declare` methods; the order of statements is immaterial. Dividing statements into multiple methods improves the readability of the program and also allows the use of Ruby's method overriding and inheritance features: because a Bloom class is just a stylized Ruby class, any of the methods in a Bloom class can be overridden by a subclass. We expand upon this idea next.

3.4 Modules and Interfaces

Conventional wisdom in certain quarters says that rule-based languages are untenable for large programs that evolve over time, since the interactions among rules become too difficult to understand. Bloom addresses this concern in two different ways. First, unlike many prior rule-based languages, Bloom is purely declarative; this avoids forcing the programmer to reason about the interaction between declarative statements and imperative constructs. Second, Bloom borrows object-oriented features from Ruby to enable programs to be broken into small modules and to allow modules to interact with one another by exposing narrow interfaces. This aids program comprehension, because it reduces the amount of code a programmer needs to read to understand the behavior of a module.

A Bloom module is a bundle of collections and statements. Like modules in Ruby, a Bloom module can "mixin" one or more other modules via the `include` statement; mixing-in a module imports its collections and statements. A common pattern is to specify an abstract interface in one module and then use the mixin feature to specify several concrete realizations in separate modules. To support this idiom, Bloom provides a special type of collection called an **interface**. An input interface defines a place where a module accepts stimuli from the outside world (e.g., other Bloom modules). Typically, inserting a fact into an input interface results in a corresponding fact appearing (perhaps after a delay) in one of the module's output interfaces.

For example, the DeliveryProtocol module in Figure 2 defines an abstract interface for sending messages to a remote address. Clients use an implementation of this interface by inserting a fact into `pipe_in`; this represents a new message to be delivered. A corresponding fact will eventually appear in the `pipe_sent` output interface; this indicates that the delivery operation has been completed. The ReliableDelivery module of Figure 2 is one possible implementation of the abstract DeliveryProtocol interface—it uses a buffer and acknowledgment messages to delay emitting a `pipe_sent` fact until the corresponding message has been acknowledged by the remote node. Figure 18 in Appendix A contains a different implementation of the abstract DeliveryProtocol. A client program that is indifferent to the details of message delivery can simply interact with the abstract DeliveryProtocol; the particular implementation of this protocol can be chosen independently.

A common requirement is for one module to "override" some of the statements in a module that it mixes in. For example, an OrderedDelivery module might want to reuse the functionality provided by ReliableDelivery but prevent a message with sequence number x from being delivered until all messages with sequence numbers $< x$ have been acknowledged. To support this pattern, Bloom allows an interface defined in another module to be overridden simply by redeclaring it. Internally, both of these redundantly-named interfaces exist in the namespace of the module that declared them, but they

[2]Note that many of these methods are provided by the standard Ruby `Enumerable` module, which `BudCollection` imports.

only need to be referenced by a fully qualified name if their use is otherwise ambiguous. If an input interface appears in the lhs of a statement in a module that declared the interface, it is rewritten to reference the interface with the same name in a mixed-in class, because a module cannot insert into its own input interface. The same is the case for output interfaces appearing in the rhs of statements. This feature allows programmers to reuse existing modules and interpose additional logic in a style reminiscent of superclass invocation in object-oriented languages. We provide an example of interface overriding in Section 4.3.

3.5 Bud Implementation

Bud is intended to be a lightweight rapid prototype of Bloom: a first effort at embodying the Dedalus logic in a syntax familiar to programmers. Bud consists of less than 2400 lines of Ruby code, developed as a part-time effort over the course of a semester.

A Bud program is just a Ruby class definition. To make it operational, a small amount of imperative Ruby code is needed to create an instance of the class and invoke the Bud `run` method. This imperative code can then be launched on as many nodes as desired (e.g., via the popular Capistrano package for Ruby deployments). As an alternative to the `run` method, the Bud class also provides a `tick` method that can be used to force evaluation of a single timestep; this is useful for debugging Bloom code with standard Ruby debugging tools or for executing a Bud program that is intended as a "one-shot" query.

Because Bud is pure Ruby, some programmers may choose to embed it as a domain-specific language (DSL) within traditional imperative Ruby code. In fact, nothing prevents a subclass of Bud from having both Bloom code in `declare` methods and imperative code in traditional Ruby methods. This is a fairly common usage model for many DSLs. A mixture of declarative Bloom methods and imperative Ruby allows the full range of existing Ruby code—including the extensive RubyGems repositories—to be combined with checkable distributed Bloom programs. The analyses we describe in the remaining sections still apply in these cases; the imperative Ruby code interacts with the Bloom logic in the same way as any external agent sending and receiving network messages.

4. CASE STUDY: KEY-VALUE STORE

In this section, we present two variants of a key-value store (KVS) implemented using Bloom.[3] We begin with an abstract protocol that any key-value store will satisfy, and then provide both single-node and replicated implementations of this protocol. We then introduce a graphical visualization of the dataflow in a Bloom program and use this visualization to reason about the *points of order* in our programs: places where additional coordination may be required to guarantee consistent results.

4.1 Abstract Key-Value Store Protocol

Figure 4 specifies a protocol for interacting with an abstract key-value store. The protocol comprises two input interfaces (representing attempts to insert and fetch items from the store) and a single output interface (which represents the outcome of a fetch operation). To use an implementation of this protocol, a Bloom program can store key-value pairs by inserting facts into `kvput`. To retrieve the value associated with a key, the client program inserts a fact into `kvget` and looks for a corresponding response tuple in `kvget_response`. For both put and get operations, the client must supply a unique request identifier (`reqid`) to differentiate tuples in the event of multiple concurrent requests.

[3]The complete source code for both of the case studies presented in this paper can be found at `http://boom.cs.berkeley.edu/cidr11/`.

```
0 module KVSProtocol
1   def state
2     interface input, :kvput,
3       ['client', 'key', 'reqid'], ['value']
4     interface input, :kvget, ['reqid'], ['key']
5     interface output, :kvget_response,
6       ['reqid'], ['key', 'value']
7   end
8 end
```

Figure 4: Abstract key-value store protocol.

```
0  module BasicKVS
1    include KVSProtocol
2
3    def state
4      table :kvstate, ['key'], ['value']
5    end
6
7    declare
8    def do_put
9      kvstate <+ kvput.map{|p| [p.key, p.value]}
10     prev = join [kvstate, kvput], [kvstate.key, kvput.key]
11     kvstate <- prev.map{|b, p| b}
12   end
13
14   declare
15   def do_get
16     getj = join [kvget, kvstate], [kvget.key, kvstate.key]
17     kvget_response <= getj.map do |g, t|
18       [g.reqid, t.key, t.value]
19     end
20   end
21 end
```

Figure 5: Single-node key-value store implementation.

A module which uses a key-value store but is indifferent to the specifics of the implementation may simply mixin the abstract protocol and postpone committing to a particular implementation until runtime. As we will see shortly, an implementation of the KVSProtocol is a collection of Bloom statements that read tuples from the protocol's input interfaces and send results to the output interface.

4.2 Single-Node Key-Value Store

Figure 5 contains a single-node implementation of the abstract key-value store protocol. Key-value pairs are stored in a persistent table called `kvstate` (line 4). When a `kvput` tuple is received, its key-value pair is stored in `kvstate` at the next timestep (line 9). If the given key already exists in `kvstate`, we want to replace the key's old value. This is done by joining `kvput` against the current version of `kvstate` (line 10). If a matching tuple is found, the old key-value pair is removed from `kvstate` at the beginning of the next timestep (line 11). Note that we also insert the new key-value pair into `kvstate` in the next timestep (line 9); hence, an overwriting update is implemented as an atomic deletion and insertion.

4.3 Replicated Key-Value Store

Next, we extend the basic key-value store implementation to support replication (Figure 6). To communicate between replicas, we use a simple multicast library implemented in Bloom; the source code for this library can be found in Appendix A. To send a multicast, a program inserts a fact into `send_mcast`; a corresponding fact appears in `mcast_done` when the multicast is complete. The multicast library also exports the membership of the multicast group in a table called `members`.

Our replicated key-value store is implemented on top of the single-node key-value store described in the previous section. When a new key is inserted by a client, we multicast the insertion to the other

```
0  module ReplicatedKVS
1    include BasicKVS
2    include MulticastProtocol

4    def state
5      interface input, :kvput,
6        ['client', 'key', 'reqid'], ['value']
7    end

9    declare
10   def replicate
11     send_mcast <= kvput.map do |k|
12       unless members.include? [k.client]
13         [k.reqid, [@local_addr, k.key, k.reqid, k.value]]
14       end
15     end
16   end

18   declare
19   def apply_put
20     kvput <= mcast_done.map{|m| m.payload}

22     kvput <= pipe_chan.map do |d|
23       if d.payload.fetch(1) != @local_addr
24         d.payload
25       end
26     end
27   end
28 end
```

Figure 6: Replicated key-value store implementation.

```
0  class RealizedReplicatedKVS < Bud
1    include ReplicatedKVS
2    include SimpleMulticast
3    include BestEffortDelivery
4  end

6  kvs = RealizedReplicatedKVS.new("localhost", 12345)
7  kvs.run
```

Figure 7: A fully specified key-value store program.

replicas (lines 11–15). To avoid repeated multicasts of the same inserted key, we avoid multicasting updates we receive from another replica (line 12). We apply an update to our local `kvstate` table in two cases: (1) if a multicast succeeds at the node that originated it (line 20) (2) whenever a multicast is received by a peer replica (lines 22–26). Note that `@local_addr` is a Ruby instance variable defined by Bud that contains the network address of the current Bud instance.

In Figure 6 ReplicatedKVS wants to "intercept" `kvput` events from clients, and only apply them to the underlying BasicKVS module when certain conditions are met. To achieve this, we "override" the declaration of the `kvput` input interface as discussed in Section 3.4 (lines 5–6). In ReplicatedKVS, references to `kvput` appearing in the lhs of statements are resolved to the `kvput` provided by BasicKVS, while references in the rhs of statements resolve to the local `kvput`. As described in Section 3.4, this is unambiguous because a module cannot insert into its own input or read from its own output interfaces.

Figure 7 combines ReplicatedKVS with a concrete implementation of MulticastProtocol and DeliveryProtocol. The resulting class, a subclass of Bud, may be instantiated and run as shown in lines 6 and 7.

4.4 Predicate Dependency Graphs

Now that we have introduced two concrete implementations of the abstract key-value store protocol, we turn to analyzing the properties of these programs. We begin by describing the graphical dataflow representation used by our analysis. In the following section, we discuss the dataflow graphs generated for the two key-value store implementations.

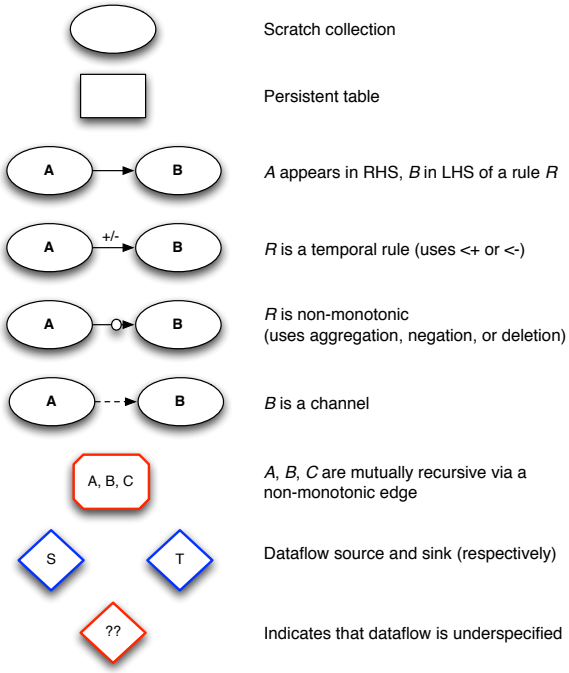


Figure 8: Visual analysis legend.

A Bloom program may be viewed as a dataflow graph with external input interfaces as sources, external output interfaces as sinks, collections as internal nodes, and rules as edges. This graph represents the dependencies between the collections in a program and is generated automatically by the Bud interpreter. Figure 8 contains a list of the different symbols and annotations in the graphical visualization; we provide a brief summary below.

Each node in the graph is either a collection or a cluster of collections; tables are shown as rectangles, ephemeral collections (scratch, periodic and channel) are depicted as ovals, and clusters (described below) as octagons. A directed edge from node A to node B indicates that B appears in the lhs of a Bloom statement that references A in the rhs, either directly or through a join expression. An edge is annotated based on the operator symbol in the statement. If the statement uses the <+ or <- operators, the edge is marked with "+/−". This indicates that facts traversing the edge "spend" a timestep to move from the rhs to the lhs. Similarly, if the statement uses the <~ operator, the edge is a dashed line—this indicates that facts from the rhs appear at the lhs at a non-deterministic future time. If the statement involves a non-monotonic operation (aggregation, negation, or deletion via the <- operator), then the edge is marked with a white circle. To make the visualizations more readable, any strongly connected component marked with both a circle and a +/− edge is collapsed into an octagonal "temporal cluster," which can be viewed abstractly as a single, non-monotonic node in the dataflow. Any non-monotonic edge in the graph is a *point of order*, as are all edges incident to a temporal cluster, including their implicit self-edge.

4.5 Analysis

Figure 9 presents a visual representation of the abstract key-value store protocol. Naturally, the abstract protocol does not specify a connection between the input and output events; this is indicated in the diagram by the red diamond labeled with "??", denoting an underspecified dataflow. A concrete realization of the key-value

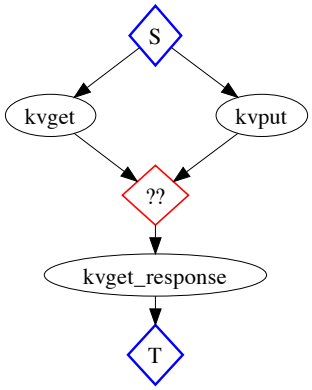


Figure 9: Visualization of the abstract key-value store protocol.

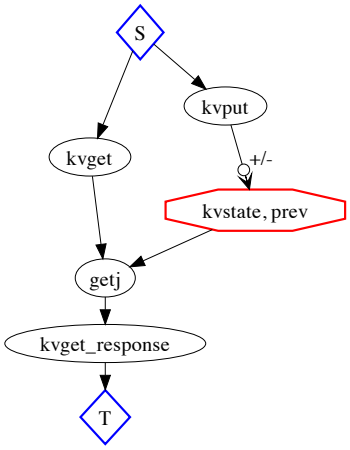


Figure 10: Visualization of the single-node key-value store.

store protocol must, at minimum, supply a dataflow that connects an input interface to an output interface.

Figure 10 shows the visual analysis of the single-node KVS implementation, which supplies a concrete dataflow for the unspecified component in the previous graph. `kvstate` and `prev` are collapsed into a red octagon because they are part of a strongly connected component in the graph with both negative and temporal edges. Any data flowing from `kvput` to the sink must cross at least one non-monotonic point of order (at ingress to the octagon) and possibly an arbitrary number of them (by traversing the dependency cycle collapsed into the octagon), and any path from `kvget` to the sink must join state potentially affected by non-monotonicity (because `kvstate` is used to derive `kvget_response`).

Reviewing the code in Figure 5, we see the source of the non-monotonicity. The contents of `kvstate` may be defined via a "destructive" update that combines the previous state and the current input from `kvput` (lines 9–11 of Figure 5). Hence the contents of `kvstate` may depend on the order of arrival of `kvput` tuples.

5. CASE STUDY: SHOPPING CART

In this section, we develop two different designs for a distributed shopping-cart service in Bloom. In a shopping cart system, clients add and remove items from their shopping cart. To provide fault tolerance and persistence, the content of the cart is stored by a collection of server replicas. Once a client has finished shopping, they perform a "checkout" request, which returns the final state of their cart.

After presenting the abstract shopping cart protocol and a simple client program, we implement a "destructive," state-modifying shopping cart service that uses the key-value store introduced in Section 4. Second, we illustrate a "disorderly" cart that accumulates updates in a set-wise fashion, summarizing updates at checkout into a final result. These two different designs illustrate our analysis tools and the way they inform design decisions for distributed programming.

```
0  module CartProtocol
1    def state
2      channel :action_msg,
3        ['@server', 'client', 'session', 'reqid'],
4        ['item', 'action']
5      channel :checkout_msg,
6        ['@server', 'client', 'session', 'reqid']
7      channel :response_msg,
8        ['@client', 'server', 'session'], ['contents']
9    end
10 end

12 module CartClientProtocol
13   def state
14     interface input, :client_action,
15       ['server', 'session', 'reqid'], ['item', 'action']
16     interface input, :client_checkout,
17       ['server', 'session', 'reqid']
18     interface output, :client_response,
19       ['client', 'server', 'session'], ['contents']
20   end
21 end
```

Figure 11: Abstract shopping cart protocol.

```
0  module CartClient
1    include CartProtocol
2    include CartClientProtocol

4    declare
5    def client
6      action_msg <~ client_action.map do |a|
7        [a.server, @local_addr, a.session, a.reqid, a.item, a.action]
8      end
9      checkout_msg <~ client_checkout.map do |a|
10       [a.server, @local_addr, a.session, a.reqid]
11     end
12     client_response <= response_msg
13   end
14 end
```

Figure 12: Shopping cart client implementation.

5.1 Shopping Cart Client

An abstract shopping cart protocol is presented in Figure 11. Figure 12 contains a simple shopping cart client program: it takes client operations (represented as `client_action` and `client_checkout` facts) and sends them to the shopping cart service using the CartProtocol. We omit logic for clients to choose a cart server replica; this can be based on simple policies like round-robin or random selection, or via more explicit load balancing.

5.2 "Destructive" Shopping Cart Service

We begin with a shopping cart service built on a key-value store. Each cart is a `(key,value)` pair, where `key` is a unique session identifier and `value` is an object containing the session's state, including a Ruby array that holds the items currently in the cart. Adding or deleting items from the cart result in "destructive" updates: the value associated with the key is replaced by a new value

```
0  module DestructiveCart
1    include CartProtocol
2    include KVSProtocol

4    declare
5    def do_action
6      kvget <= action_msg.map{|a| [a.reqid, a.key]}

8      kvput <= action_msg.map do |a|
9        if a.action == "A"
10         unless kvget_response.map{|b| b.key}.include? a.session
11           [a.server, a.client, a.session, a.reqid, [a.item]]
12         end
13       end
14     end

16     old_state = join [kvget_response, action_msg],
17                      [kvget_response.key, action_msg.session]
18     kvput <= old_state.map do |b, a|
19       if a.action == "A"
20         [a.server, a.client, a.session,
21          a.reqid, b.value.push(a.item)]
22       elsif a.action == "D"
23         [a.server, a.client, a.session,
24          a.reqid, delete_one(b.value, a.item)]
25       end
26     end
27   end

29   declare
30   def do_checkout
31     kvget <= checkout_msg.map{|c| [c.reqid, c.session]}
32     lookup = join [kvget_response, checkout_msg],
33                   [kvget_response.key, checkout_msg.session]
34     response_msg <~ lookup.map do |r, c|
35       [c.client, c.server, c.session, r.value]
36     end
37   end
38 end
```

Figure 13: Destructive cart implementation.

that reflects the effect of the update. Deletion requests are ignored if the item they refer to does not exist in the cart.

Figure 13 shows the Bloom code for this design. The `kvput` collection is provided by the abstract KVSProtocol described in Section 4. Our shopping cart service would work with any concrete realization of the KVSProtocol; we will choose to use the replicated key-value store (Section 4.3) to provide fault-tolerance.

When client actions arrive from the CartClient, the cart service checks to see if there is a record in the key-value store associated with the client's session. If no record is found (i.e., this is the first operation for a new session), then lines 9–13 generate an entry for the new session in `kvstate`. Otherwise, the join conditions in line 17 are satisfied and lines 19–25 "replace" the value in the key-value store with an updated set of items for this session; this uses the built-in overwriting capability provided by the key-value store. When a `checkout_msg` appears at a server replica, the key-value store is queried to retrieve the cart state associated with the given session (lines 31–35), and the results are returned to the client.

5.3 "Disorderly" Shopping Cart Service

Figure 14 shows an alternative shopping cart implementation, in which updates are monotonically accumulated in a set, and summed up only at checkout. Lines 12–14 insert client updates into the persistent table `cart_action`. Lines 15–17 define `action_cnt` as an aggregate over `cart_action`, in the style of an SQL `GROUP BY` statement: for each item associated with a cart, we separately count the number of times it was added and the number of times it was deleted. Lines 22–27 ensure that when a `checkout_msg` tuple arrives, `status` contains a record for every added item for which there was no corresponding deletion in the session. Lines 29–36

```
0  module DisorderlyCart
1    include CartProtocol

3    def state
4      table :cart_action, ['session', 'item', 'action', 'reqid']
5      table :action_cnt, ['session', 'item', 'action'], ['cnt']
6      scratch :status, ['server', 'client', 'session', 'item'],
7                       ['cnt']
8    end

10   declare
11   def do_action
12     cart_action <= action_msg.map do |c|
13       [c.session, c.item, c.action, c.reqid]
14     end
15     action_cnt <= cart_action.group(
16       [cart_action.session, cart_action.item, cart_action.action],
17       count(cart_action.reqid))
18   end

20   declare
21   def do_checkout
22     del_items = action_cnt.map{|a| a.item if a.action == "Del"}
23     status <= join([action_cnt, checkout_msg]).map do |a, c|
24       if a.action == "Add" and not del_items.include? a.item
25         [c.client, c.server, a.session, a.item, a.cnt]
26       end
27     end

29     status <= join([action_cnt, action_cnt,
30                     checkout_msg]).map do |a1, a2, c|
31       if a1.session == a2.session and a1.item == a2.item and
32          a1.session == c.session and
33          a1.action == "A" and a2.action == "D"
34         [c.client, c.server, c.session, a1.item, a1.cnt - a2.cnt]
35       end
36     end

38    response_msg <~ status.group(
39       [status.client, status.server, status.session],
40       accum(status.cnt.times.map{status.item}))
41   end
42 end
```

Figure 14: Disorderly cart implementation.

additionally define `status` as the 3-way join of the `checkout_msg` message and two copies of `action_cnt`—one corresponding to additions and one to deletions. Thus, for each item, `status` contains its final quantity: the difference between the number of additions and deletions (line 34), or simply the number of additions if there are no deletions (line 25). Upon the appearance of a `checkout_msg`, the replica returns a `response_msg` to the client containing the final quantity (lines 38–40). Because the CartClient expects the cart to be returned as an array of items on checkout, we use the `accum` aggregate function to nest the set of items into an array.

5.4 Analysis

Figure 15 presents the analysis of the "destructive" shopping cart variant. Note that because all dependencies are analyzed, collections defined in mixins but not referenced in the code sample (e.g., `pipe_chan`, `member`) also appear in the graph. Although there is no syntactic non-monotonicity in Figure 13, the underlying key-value store uses the non-monotonic `<-` operator to model updateable state. Thus, while the details of the implementation are encapsulated by the key-value store's abstract interface, its points of order resurface in the full-program analysis. Figure 15 indicates that there are points of order between `action_msg`, `member`, and the temporal cluster. This figure also tells the (sad!) story of how we could ensure consistency of the destructive cart implementation: introduce coordination between client and server—and between the chosen server and all its replicas—for *every client action or kvput update*. The programmer can achieve this coordination by supplying a "reliable"

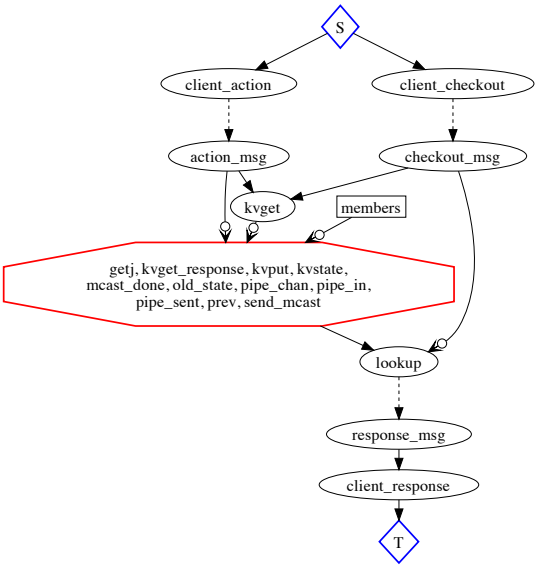


Figure 15: Visualization of the destructive cart program.

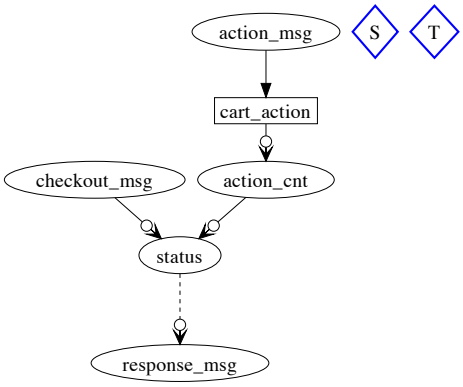


Figure 16: Visualization of the core logic for the disorderly cart.

implementation of multicast that awaits acknowledgements from all replicas before reporting completion: this fine-grained coordination is akin to "eager replication" [9]. Unfortunately, it would incur the latency of a round of messages per server per client update, decrease throughput, and reduce availability in the face of replica failures.

Because we only care about the *set* of elements contained in the value array and not its order, we might be tempted to argue that the shopping cart application is eventually consistent when asynchronously updated and forego the coordination logic. Unfortunately, such informal reasoning can hide serious bugs. For example, consider what would happen if a delete action for an item arrived at some replica before any addition of that item: the delete would be ignored, leading to inconsistencies between replicas.

A happier story emerges from our analysis of the disorderly cart service. Figure 16 shows a visualization of the core logic of the disorderly cart module presented in Figure 14. This program is not complete: its inputs and outputs are channels rather than interfaces, so the dataflow from source to sink is not completed. To complete this program, we must mixin code that connects input and output interfaces to `action_msg`, `checkout_msg`, and `response_msg`, as

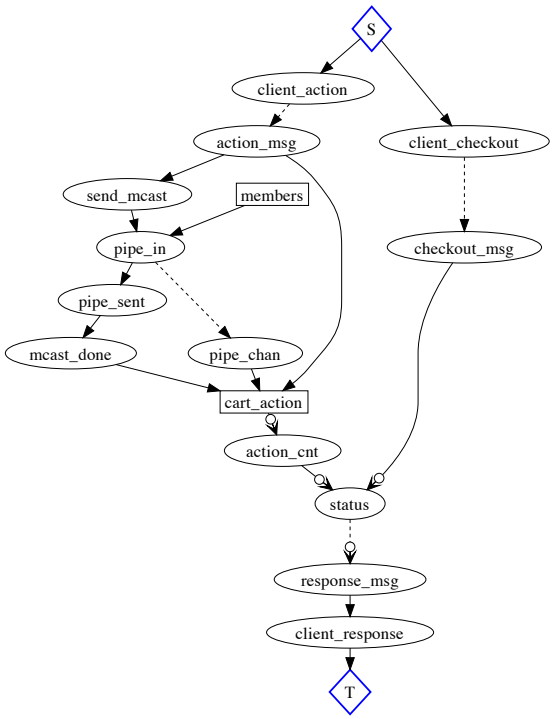


Figure 17: Visualization of the complete disorderly cart program.

the CartClient does (Figure 12). Note that the disorderly cart has points of order on all paths but there are no cycles.

Figure 17 shows the analysis for a complete implementation that mixes in both the client code and logic to replicate the `cart_action` table via best-effort multicast (see Figure 20 in Appendix A for the corresponding source code). Note that communication (via `action_msg`) between client and server—and among server replicas—crosses no points of order, so all the communication related to shopping actions converges to the same final state without coordination. However, there are points of order upon the appearance of `checkout_msg` messages, which must be joined with an `action_cnt` aggregate over the set of updates. Additionally, using the `accum` aggregate adds a point of order to the end of the dataflow, between `status` and `response_msg`. Although the accumulation of shopping actions is monotonic, summarization of the cart state requires us to ensure that there will be no further cart actions.

Comparing Figure 15 and Figure 17, we can see that the disorderly cart requires less coordination than the destructive cart: to ensure that the response to the client is deterministic and consistently replicated, we need to coordinate once per *session* (at checkout), rather than once per shopping action. This is analogous to the desired behavior in practice [13].

5.5 Discussion

Strictly monotonic programs are rare in practice, so adding some amount of coordination is often required to ensure consistency. In this running example we studied two candidate implementations of a simple distributed application with the aid of our program analysis. Both programs have points of order, but the analysis tool helped us reason about their relative coordination costs. Deciding that the disorderly approach is "better" required us to apply domain

knowledge: checkout is a coarser-grained coordination point than cart actions and their replication.

By providing the programmer with a set of abstractions that are predominantly order-independent, Bloom encourages a style of programming that minimizes coordination requirements. But as we see in the destructive cart program, it is nonetheless possible to use Bloom to write code in an imperative, order-sensitive style. Our analysis tools provide assistance in this regard. Given a particular implementation with points of order, Bloom's dataflow analysis can help a developer iteratively refine their program—either to "push back" the points to as late as possible in the dataflow, as we did in this example, or to "localize" points of order by moving them to locations in the program's dataflow where the coordination can be implemented on individual nodes without communication.

6. TOLERATING INCONSISTENCY

In the previous section we showed how to identify points of order: code locations that are sensitive to non-deterministic input ordering. We then demonstrated how to resolve the non-determinism by introducing coordination. However, in many cases adding additional coordination is undesirable due to concerns like latency and availability. In these cases, Bloom's point-of-order analysis can assist programmers with the task of *tolerating inconsistency*, rather than resolving it via coordination. A notable example of how to manage inconsistency is presented by Helland and Campbell, who reflect on their experience programming with patterns of "memories, guesses and apologies" [13]. We provide a sketch here of ideas for converting these patterns into developer tools in Bloom.

"Guesses"—facts that may not be true—may arise at the inputs to a program, e.g., from noisy sensors or untrusted software or users. But Helland and Campbell's use of the term corresponds in our analysis to unresolved points of order: non-monotonic logic that makes decisions without full knowledge of its input sets. We can rewrite the schemas of Bloom collections to include an additional attribute marking each fact as a "guarantee" or "guess," and automatically augment user code to propagate those labels through program logic in the manner of "taint checking" in program security [22, 25]. Moreover, by identifying unresolved points of order, we can identify when program logic derives "guesses" from "guarantees," and rewrite user code to label data appropriately. By rewriting programs to log guesses that cross interface boundaries, we can also implement Helland and Campbell's idea of "memories": a log of guesses that were sent outside the system.

Most of these patterns can be implemented as automatic program rewrites. We envision building a system that facilitates running low-latency, "guess"-driven decision making in the foreground, and expensive but consistent logic as a background process. When the background process detects an inconsistency in the results produced by the foreground system (e.g., because a "guess" turns out to be mistaken), it can then take corrective action by generating an "apology." Importantly, both of these subsystems are implementations of the same high-level design, except with different consistency and coordination requirements; hence, it should be possible to synthesize both variants of the program from the same source code. Throughout this process—making calculated "guesses," storing appropriate "memories," and generating the necessary "apologies"—we see significant opportunities to build scaffolding and tool support to lighten the burden on the programmer.

Finally, we hope to provide analysis techniques that can prove the consistency of the high-level workflow: i.e., prove that any combination of user behavior, background guess resolution, and apology logic will eventually lead to a consistent resolution of the business rules at both the user and system sides.

7. RELATED WORK

Systems with loose consistency requirements have been explored in depth by both the systems and database management communities (e.g., [6, 8, 9, 24]); we do not attempt to provide an exhaustive survey here. The shopping cart case study in Section 5 was motivated by the Amazon Dynamo paper [11], as well as the related discussion by Helland and Campbell [13].

The Bloom language is inspired by earlier work that attempts to integrate databases and programming languages. This includes early research such as Gem [27] and more recent object-relational mapping layers such as Ruby on Rails. Unlike these efforts, Bloom is targeted at the development of both distributed infrastructure and distributed applications, so it does not make any assumptions about the presence of a database system "underneath". Given our prototype implementation in Ruby, it is tempting to integrate Bud with Rails; we have left this for future work.

There is a long history of attempts to design programming languages more suitable to parallel and distributed systems; for example, Argus [15] and Linda [7]. Again, we do not hope to survey that literature here. More pragmatically, Erlang is an oft-cited choice for distributed programming in recent years. Erlang's features and design style encourage the use of asynchronous lightweight "actors." As mentioned previously, we did a simple Bloom prototype DSL in Erlang (which we cannot help but call "Bloomerlang"), and there is a natural correspondence between Bloom-style distributed rules and Erlang actors. However there is no requirement for Erlang programs to be written in the disorderly style of Bloom. It is not obvious that typical Erlang programs are significantly more amenable to a useful points-of-order analysis than programs written in any other functional language. For example, ordered lists are basic constructs in functional languages, and without program annotation or deeper analysis than we need to do in Bloom, any code that modifies lists would need be marked as a point of order, much like our destructive shopping cart. We believe that Bloom's "disorderly by default" style encourages order-independent programming, and we know that its roots in database theory helped produce a simple but useful program analysis technique. While we would be happy to see the analysis "ported" to other distributed programming environments, it may be that design patterns using Bloom-esque disorderly programming are the natural way to achieve this.

Our work on Bloom bears a resemblance to the Reactor language [5]. Both languages target distributed programming and are grounded in Datalog. Like many other rule languages including our earlier work on Overlog, Reactor updates mutable state in an operational step "outside Datalog" after each fixpoint computation. By contrast, Bloom is purely declarative: following Dedalus, it models updates as the logical derivation of immutable "versions" of collections over time. While Bloom uses a syntax inspired by object-oriented languages, Reactor takes a more explicitly agent-oriented approach. Reactor also includes synchronous coupling between agents as a primitive; we have opted to only include asynchronous communication as a language primitive and to provide synchronous coordination between nodes as a library.

Another recent language related to our work is Coherence [4], which also embraces "disorderly" programming. Unlike Bloom, Coherence is not targeted at distributed computing and is not based on logic programming.

8. CONCLUSION AND FUTURE WORK

In this paper we make three main contributions. First, we present the CALM principle, which connects the notion of eventual consistency in distributed programming to theoretical foundations in

database theory. Second, we show that we can bring that theory to bear on the practice of software development via "disorderly" programming patterns, complemented with automatic analysis techniques for identifying and managing a program's points of order in a principled way. Finally, we present our Bloom prototype as an example of a practically-minded disorderly and declarative programming language, with an initial implementation as a domain-specific language within Ruby.

We plan to extend the work described in this paper in several directions. First, we are building a more mature Bloom language environment, including a library of modules for distributed computing. We intend to compose those modules to implement a number of variants of distributed systems. The design of Bloom itself was motivated by our experience implementing scalable services and protocols in Overlog [1, 2], and this practice of system/language co-design continues to be part of our approach. Second, we hope to expand our suite of analysis techniques to address additional important properties in distributed systems, including idempotency and invertability of interfaces. Third, we are hopeful that the logic foundation of Bloom will enable us to develop better tools and techniques for the debugging and systematic testing of distributed systems under failure and security attacks, perhaps drawing on recent work on this topic [10, 17]. Finally, we are working to formally tighten our ideas connecting non-monotonic logic, distributed coordination, and consistency of distributed programs [14].

Acknowledgments

We would like to thank Ras Bodík, Kuang Chen, Haryadi Gunawi, Dmitriy Ryaboy, Russell Sears, and the anonymous reviewers for their helpful comments. This work was supported by NSF grants 0917349, 0803690, 0722077, 0713661 and 0435496, Air Force Office of Scientific Research award 22178970-41070-F, the Natural Sciences and Engineering Research Council of Canada, and gifts from Yahoo Research, IBM Research and Microsoft Research.

9. REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[2] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I Do Declare: Consensus in a Logic Language. *SIGOPS Oper. Syst. Rev.*, 43:25–30, January 2010.

[3] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. In *Proc. Datalog 2.0 Workshop (to appear)*, 2011.

[4] J. Edwards. Coherent Reaction. In *OOPSLA*, 2009.

[5] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. *Theoretical Computer Science*, 410(2-3), February 2009.

[6] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[7] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[8] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, 1996.

[10] H. S. Gunawi et al. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI (to appear)*, 2011.

[11] D. Hastorun et al. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[12] P. Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*, 2007.

[13] P. Helland and D. Campbell. Building on Quicksand. In *CIDR*, 2009.

[14] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[15] B. Liskov. Distributed programming in Argus. *CACM*, 31:300–312, 1988.

[16] B. T. Loo et al. Implementing Declarative Overlays. In *SOSP*, 2005.

[17] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, 2010.

[18] D. Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55, 2008.

[19] T. C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[20] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *PODS*, 1990.

[21] K. A. Ross. A syntactic stratification condition using constraints. In *International Symposium on Logic Programming*, pages 76–90, 1994.

[22] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications*, 21(1):5–19, 2003.

[23] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, 1986.

[24] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[25] S. Vandebogart et al. Labels and Event Processes in the Asbestos Operating System. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[26] W. Vogels. Eventually Consistent. *CACM*, 52(1):40–44, 2009.

[27] C. Zaniolo. The database language GEM. In *SIGMOD*, 1983.

APPENDIX

A. ADDITIONAL SOURCE CODE

Figures 18, 19 and 20 contain the remainder of the Bloom code used in this paper: best-effort protocols for unicast and multicast messaging, and the complete program for the replicated disorderly cart described in Section 5.

```
0  module BestEffortDelivery
1    include DeliveryProtocol

3    def state
4      channel :pipe_chan,
5        ['@dst', 'src', 'ident'], ['payload']
6    end

8    declare
9    def snd
10     pipe_chan <~ pipe_in
11   end

13   declare
14   def done
15     pipe_sent <= pipe_in
16   end
17 end
```

Figure 18: Best-effort unicast messaging in Bloom.

```
0  module MulticastProtocol
1    def state
2      table :members, ['peer']
3      interface input, :send_mcast, ['ident'], ['payload']
4      interface output, :mcast_done, ['ident'], ['payload']
5    end
6  end

8  module SimpleMulticast
9    include MulticastProtocol
10   include DeliveryProtocol

12   declare
13   def snd_mcast
14     pipe_in <= join([send_mcast, members]).map do |s, m|
15       [m.peer, @local_addr, s.ident, s.payload]
16     end
17   end

19   declare
20   def done_mcast
21     mcast_done <= pipe_sent.map{|p| [p.ident, p.payload]}
22   end
23 end
```

Figure 19: A simple unreliable multicast library in Bloom.

```
0  class ReplicatedDisorderlyCart < Bud
1    include DisorderlyCart
2    include SimpleMulticast
3    include BestEffortDelivery

5    declare
6    def replicate
7      send_mcast <= action_msg.map do |a|
8        [a.reqid, [a.session, a.item, a.action, a.reqid]]
9      end
10     cart_action <= pipe_chan.map{|c| c.payload }
11   end
12 end
```

Figure 20: The complete disorderly cart program.

Starfish: A Self-tuning System for Big Data Analytics

Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong,
Fatma Bilgen Cetin, Shivnath Babu
Department of Computer Science
Duke University

ABSTRACT

Timely and cost-effective analytics over "Big Data" is now a key ingredient for success in many businesses, scientific and engineering disciplines, and government endeavors. The Hadoop software stack—which consists of an extensible MapReduce execution engine, pluggable distributed storage engines, and a range of procedural to declarative interfaces—is a popular choice for big data analytics. Most practitioners of big data analytics—like computational scientists, systems researchers, and business analysts—lack the expertise to tune the system to get good performance. Unfortunately, Hadoop's performance out of the box leaves much to be desired, leading to suboptimal use of resources, time, and money (in pay-as-you-go clouds). We introduce Starfish, a self-tuning system for big data analytics. Starfish builds on Hadoop while adapting to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning knobs in Hadoop. While Starfish's system architecture is guided by work on self-tuning database systems, we discuss how new analysis practices over big data pose new challenges; leading us to different design choices in Starfish.

1. INTRODUCTION

Timely and cost-effective analytics over "Big Data" has emerged as a key ingredient for success in many businesses, scientific and engineering disciplines, and government endeavors [6]. Web search engines and social networks capture and analyze every user action on their sites to improve site design, spam and fraud detection, and advertising opportunities. Powerful telescopes in astronomy, genome sequencers in biology, and particle accelerators in physics are putting massive amounts of data into the hands of scientists. Key scientific breakthroughs are expected to come from computational analysis of such data. Many basic and applied science disciplines now have computational subareas, e.g., computational biology, computational economics, and computational journalism.

Cohen et al. recently coined the acronym *MAD*—for *Magnetism*, *Agility*, and *Depth*—to express the features that users expect from a system for big data analytics [6].

Magnetism: A magnetic system attracts all sources of data irrespective of issues like possible presence of outliers, unknown schema or lack of structure, and missing values that keep many useful data sources out of conventional data warehouses.

Agility: An agile system adapts in sync with rapid data evolution.

Depth: A deep system supports analytics needs that go far beyond conventional rollups and drilldowns to complex statistical and machine-learning analysis.

Hadoop is a MAD system that is becoming popular for big data analytics. An entire ecosystem of tools is being developed around Hadoop. Figure 1 shows a summary of the Hadoop software stack in wide use today. Hadoop itself has two primary components: a MapReduce execution engine and a distributed filesystem. While the Hadoop Distributed FileSystem (HDFS) is used predominantly as the distributed filesystem in Hadoop, other filesystems like Amazon S3 are also supported. Analytics with Hadoop involves loading data as files into the distributed filesystem, and then running parallel MapReduce computations on the data.

A combination of factors contributes to Hadoop's MADness. First, copying files into the distributed filesystem is all it takes to get data into Hadoop. Second, the MapReduce methodology is to interpret data (lazily) at processing time, and not (eagerly) at loading time. These two factors contribute to Hadoop's magnetism and agility. Third, MapReduce computations in Hadoop can be expressed directly in general-purpose programming languages like Java or Python, domain-specific languages like R, or generated automatically from SQL-like declarative languages like HiveQL and Pig Latin. This coverage of the language spectrum makes Hadoop well suited for deep analytics. Finally, an unheralded aspect of Hadoop is its extensibility, i.e., the ease with which many of Hadoop's core components like the scheduler, storage subsystem, input/output data formats, data partitioner, compression algorithms, caching layer, and monitoring can be customized or replaced.

Getting desired performance from a MAD system can be a non-trivial exercise. The practitioners of big data analytics like data analysts, computational scientists, and systems researchers usually lack the expertise to tune system internals. Such users would rather use a system that can tune itself and provide good performance automatically. Unfortunately, the same properties that make Hadoop MAD pose new challenges in the path to self-tuning:

- *Data opacity until processing:* The magnetism and agility that comes with interpreting data only at processing time poses the difficulty that even the schema may be unknown until the point when an analysis job has to be run on the data.
- *File-based processing:* Input data for a MapReduce job may be stored as few large files, millions of small files, or anything in between. Such uncontrolled data layouts are a marked contrast to the carefully-planned layouts in database systems.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

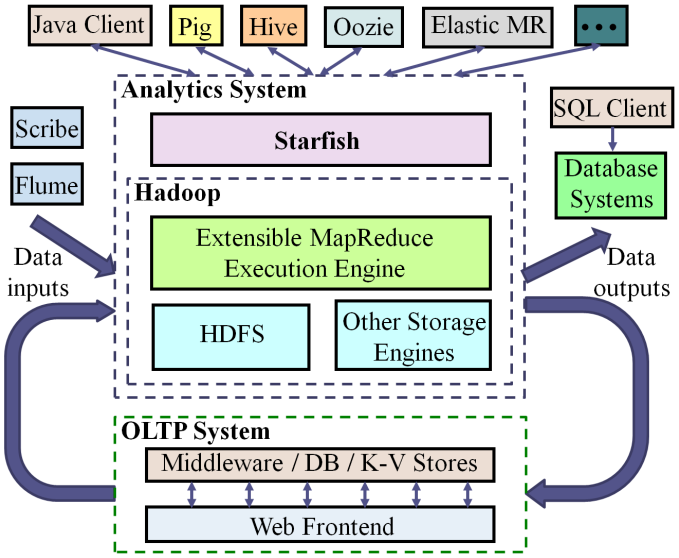


Figure 1: Starfish in the Hadoop ecosystem

- *Heavy use of programming languages:* A sizable fraction of MapReduce programs will continue to be written in programming languages like Java for performance reasons, or in languages like Python or R that a user is most comfortable with while prototyping new analysis tasks.

Traditional data warehouses are kept nonMAD by its administrators because it is easier to meet performance requirements in tightly controlled environments; a luxury we cannot afford any more [6]. To further complicate matters, three more features in addition to MAD are becoming important in analytics systems: *Data-lifecycle-awareness*, *Elasticity*, and *Robustness*. A system with all six features would be *MADDER* than current analytics systems.

Data-lifecycle-awareness: A data-lifecycle-aware system goes beyond query execution to optimize the movement, storage, and processing of big data during its entire lifecycle. The intelligence embedded in many Web sites like LinkedIn and Yahoo!—e.g., recommendation of new friends or news articles of potential interest, selection and placement of advertisements—is driven by computation-intensive analytics. A number of companies today use Hadoop for such analytics [12]. The input data for the analytics comes from dozens of different sources on user-facing systems like key-value stores, databases, and logging services (Figure 1). The data has to be moved for processing to the analytics system. After processing, the results are loaded back in near real-time to user-facing systems. Terabytes of data may go through this *cycle* per day [12]. In such settings, data-lifecycle-awareness is needed to: (i) eliminate indiscriminate data copying that causes bloated storage needs (as high as 20x if multiple departments in the company make their own copy of the data for analysis [17]); and (ii) reduce resource overheads and realize performance gains due to reuse of intermediate data or learned metadata in workflows that are part of the cycle [8].

Elasticity: An elastic system adjusts its resource usage and operational costs to the workload and user requirements. Services like Amazon Elastic MapReduce have created a market for pay-as-you-go analytics hosted on the cloud. Elastic MapReduce provisions and releases Hadoop clusters on demand, sparing users the hassle of cluster setup and maintenance.

Robustness: A robust system continues to provide service, possibly with graceful degradation, in the face of undesired events like hardware failures, software bugs [12], and data corruption.

1.1 Starfish: MADDER and Self-Tuning Hadoop

Hadoop has the core mechanisms to be MADDER than existing analytics systems. However, the use of most of these mechanisms has to be managed manually. Take elasticity as an example. Hadoop supports dynamic node addition as well as decommissioning of failed or surplus nodes. However, these mechanisms do not magically make Hadoop elastic because of the lack of control modules to decide (a) when to add new nodes or to drop surplus nodes, and (b) when and how to rebalance the data layout in this process.

Starfish is a MADDER and self-tuning system for analytics on big data. An important design decision we made is to build Starfish on the Hadoop stack as shown in Figure 1. (That is not to say that Starfish uses Hadoop as is.) Hadoop, as observed earlier, has useful primitives to help meet the new requirements of big data analytics. In addition, Hadoop's adoption by academic, government, and industrial organizations is growing at a fast pace.

A number of ongoing projects aim to improve Hadoop's peak performance, especially to match the query performance of parallel database systems [1, 7, 10]. Starfish has a different goal. The peak performance a manually-tuned system can achieve is not our primary concern, especially if this performance is for one of the many phases in the data lifecycle. Regular users may rarely see performance close to this peak. Starfish's goal is to enable Hadoop users and applications to get good performance automatically throughout the data lifecycle in analytics; without any need on their part to understand and manipulate the many tuning knobs available.

Section 2 gives an overview of Starfish while Sections 3–5 describe its components. The primary focus of this paper is on using experimental results to illustrate the challenges in each component and to motivate Starfish's solution approach.

2. OVERVIEW OF STARFISH

The *workload* that a Hadoop deployment runs can be considered at different levels. At the lowest level, Hadoop runs MapReduce *jobs*. A job can be generated directly from a program written in a programming language like Java or Python, or generated from a query in a higher-level language like HiveQL or Pig Latin [15], or submitted as part of a MapReduce job *workflow* by systems like Azkaban, Cascading, Elastic MapReduce, and Oozie. The execution plan generated for a HiveQL or Pig Latin query is usually a workflow. Workflows may be ad-hoc, time-driven (e.g., run every

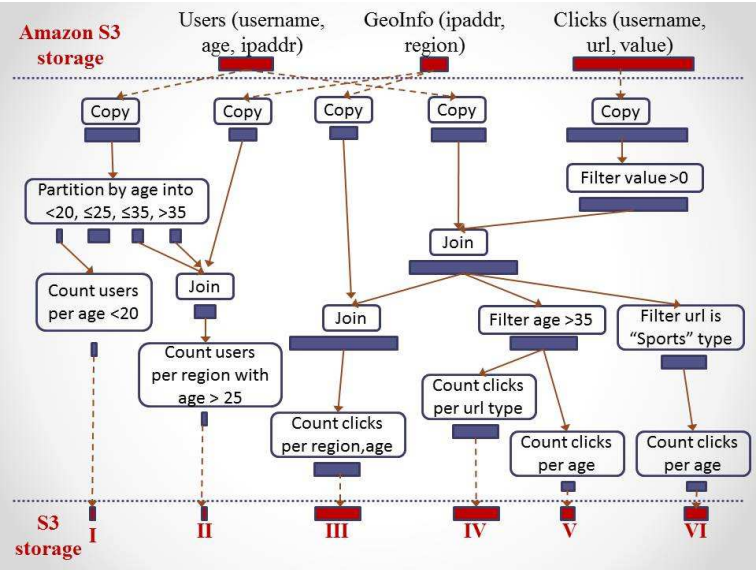


Figure 2: Example analytics workload to be run on Amazon Elastic MapReduce

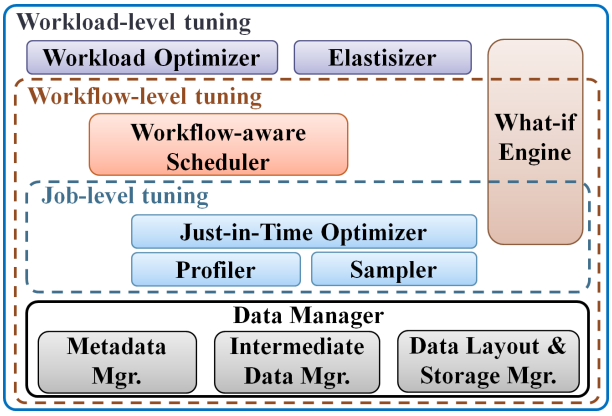


Figure 3: Components in the Starfish architecture

hour), or data-driven. Yahoo! uses data-driven workflows to generate a reconfigured preference model and an updated home-page for any user within seven minutes of a home-page click by the user.

Figure 2 is a visual representation of an example workload that a data analyst may want to run on demand or periodically using Amazon Elastic MapReduce. The input data processed by this workload resides as files on Amazon S3. The final results produced by the workload are also output to S3. The input data consists of files that are collected by a personalized Web-site like `my.yahoo.com`.

The example workload in Figure 2 consists of workflows that load the files from S3 as three datasets: Users, GeoInfo, and Clicks. The workflows process these datasets in order to generate six different results I-VI of interest to the analyst. For example, Result I in Figure 2 is a count of all users with age less than 20. For all users with age greater than 25, Result II counts the number of users per geographic region. For each workflow, one or more MapReduce jobs are generated in order to run the workflow on Amazon Elastic MapReduce or on a local Hadoop cluster. For example, notice from Figure 2 that a join of the Users and GeoInfo datasets is needed in order to generate Result II. This logical join operation can be processed using a single MapReduce job.

The tuning challenges present at each level of workload processing led us to the Starfish architecture shown in Figure 3. Broadly, the functionality of the components in this architecture can be categorized into job-level tuning, workflow-level tuning, and workload-level tuning. These components interact to provide Starfish's self-tuning capabilities.

2.1 Job-level Tuning

The behavior of a MapReduce job in Hadoop is controlled by the settings of more than 190 configuration parameters. If the user does not specify parameter settings during job submission, then default values—shipped with the system or specified by the system administrator—are used. Good settings for these parameters depend on job, data, and cluster characteristics. While only a fraction of the parameters can have significant performance impact, browsing through the Hadoop, Hive, and Pig mailing lists reveals that users often run into performance problems caused by lack of knowledge of these parameters.

Consider a user who wants to perform a join of data in the files users.txt and geoinfo.txt, and writes the Pig Latin script:

```
Users = Load 'users.txt' as (username: chararray,
            age: int, ipaddr: chararray)
GeoInfo = Load 'geoinfo.txt' as (ipaddr: chararray,
            region: chararray)
Result = Join Users by ipaddr, GeoInfo by ipaddr
```

The schema as well as properties of the data in the files could have been unknown so far. The system now has to quickly choose the join execution technique—given the limited information available so far, and from among 10+ ways to execute joins in Starfish—as well as the corresponding settings of job configuration parameters.

Starfish's *Just-in-Time Optimizer* addresses unique optimization problems like those above to automatically select efficient execution techniques for MapReduce jobs. "Just-in-time" captures the online nature of decisions forced on the optimizer by Hadoop's MADDER features. The optimizer takes the help of the *Profiler* and the *Sampler*. The Profiler uses a technique called *dynamic instrumentation* to learn performance models, called *job profiles*, for unmodified MapReduce programs written in languages like Java and Python. The Sampler collects statistics efficiently about the input, intermediate, and output *key-value spaces* of a MapReduce job. A unique feature of the Sampler is that it can sample the execution of a MapReduce job in order to enable the Profiler to collect approximate job profiles at a fraction of the full job execution cost.

2.2 Workflow-level Tuning

Workflow execution brings out some critical and unanticipated interactions between the MapReduce task scheduler and the underlying distributed filesystem. Significant performance gains are realized in parallel task scheduling by moving the computation to the data. By implication, the data layout across nodes in the cluster constrains how tasks can be scheduled in a "data-local" fashion. Distributed filesystems have their own policies on how data written to them is laid out. HDFS, for example, always writes the first replica of any block on the same node where the writer (in this case, a map or reduce task) runs. This interaction between data-local scheduling and the distributed filesystem's block placement policies can lead to an *unbalanced* data layout across nodes in the cluster during workflow execution; causing severe performance degradation as we will show in Section 4.

Efficient scheduling of a Hadoop workflow is further complicated by concerns like (a) avoiding cascading reexecution under node failure or data corruption [11], (b) ensuring power proportional computing, and (c) adapting to imbalance in load or cost of energy across geographic regions and time at the datacenter level [16]. Starfish's *Workflow-aware Scheduler* addresses such concerns in conjunction with the *What-if Engine* and the *Data Manager*. This scheduler communicates with, but operates outside, Hadoop's internal task scheduler.

2.3 Workload-level Tuning

Enterprises struggle with higher-level optimization and provisioning questions for Hadoop workloads. Given a workload consisting of a collection of workflows (like Figure 2), Starfish's *Workload Optimizer* generates an equivalent, but optimized, collection of workflows that are handed off to the Workflow-aware Scheduler for execution. Three important categories of optimization opportunities exist at the workload level:

A. *Data-flow sharing*, where a single MapReduce job performs computations for multiple and potentially different logical nodes belonging to the same or different workflows.

B. *Materialization*, where intermediate data in a workflow is stored for later reuse in the same or different workflows. Effective use of materialization has to consider the cost of materialization (both in terms of I/O overhead and storage consumption [8]) and its potential to avoid cascading reexecution of tasks under node failure or data corruption [11].

C. *Reorganization*, where new data layouts (e.g., with partitioning) and storage engines (e.g., key-value stores like HBase and databases like column-stores [1]) are chosen automatically and transparently to store intermediate data so that downstream jobs in the same or different workflows can be executed very efficiently.

While categories A, B, and C are well understood in isolation, applying them in an integrated manner to optimize MapReduce workloads poses new challenges. First, the data output from map tasks and input to reduce tasks in a job is always materialized in Hadoop in order to enable robustness to failures. This data—which today is simply deleted after the job completes—is key-value-based, sorted on the key, and partitioned using externally-specified partitioning functions. This unique form of intermediate data is available almost for free, bringing new dimensions to questions on materialization and reorganization. Second, choices for A, B, and C potentially interact among each other and with scheduling, data layout policies, as well as job configuration parameter settings. The optimizer has to be aware of such interactions.

Hadoop provisioning deals with choices like the number of nodes, node configuration, and network configuration to meet given workload requirements. Historically, such choices arose infrequently and were dealt with by system administrators. Today, users who provision Hadoop clusters on demand using services like Amazon Elastic MapReduce and Hadoop On Demand are required to make provisioning decisions on their own. Starfish's *Elastisizer* automates such decisions. The intelligence in the Elastisizer comes from a search strategy in combination with the What-if Engine that uses a mix of simulation and model-based estimation to answer what-if questions regarding workload performance on a specified cluster configuration. In the longer term, we aim to automate provisioning decisions at the level of multiple virtual and elastic Hadoop clusters hosted on a single shared Hadoop cluster to enable Hadoop *Analytics as a Service*.

2.4 Lastword: Starfish's Language for Workloads and Data

As described in Section 1.1 and illustrated in Figure 1, Starfish is built on the Hadoop stack. Starfish interposes itself between Hadoop and its clients like Pig, Hive, Oozie, and command-line interfaces to submit MapReduce jobs. These Hadoop clients will now submit workloads—which can vary from a single MapReduce job, to a workflow of MapReduce jobs, and to a collection of multiple workflows—expressed in *Lastword*[1] to Starfish. Lastword is Starfish's language to accept as well as to reason about analytics workloads.

Unlike languages like HiveQL, Pig Latin, or Java, Lastword is not a language that humans will have to interface with directly. Higher-level languages like HiveQL and Pig Latin were developed to support a diverse user community—ranging from marketing analysts and sales managers to scientists, statisticians, and systems researchers—depending on their unique analytical needs and preferences. Starfish provides language translators to automatically convert workloads specified in these higher-level languages to Lastword. A common language like Lastword allows Starfish to exploit optimization opportunities among the different workloads that run on the same Hadoop cluster.

A Starfish client submits a workload as a collection of workflows expressed in Lastword. Three types of workflows can be represented in Lastword: (a) physical workflows, which are directed graphs[2] where each node is a MapReduce job representation; (b) logical workflows, which are directed graphs where each node is a logical specification such as a select-project-join-aggregate (SPJA) or a user-defined function for performing operations like partitioning, filtering, aggregation, and transformations; and (c) hybrid workflows, where a node can be of either type.

An important feature of Lastword is its support for expressing metadata along with the tasks for execution. Workflows specified in Lastword can be annotated with metadata at the workflow level or at the node level. Such metadata is either extracted from inputs provided by users or applications, or learned automatically by Starfish. Examples of metadata include scheduling directives (e.g., whether the workflow is ad-hoc, time-driven, or data-driven), data properties (e.g., full or partial schema, samples, and histograms), data layouts (e.g., partitioning, ordering, and collocation), and runtime monitoring information (e.g., execution profiles of map and reduce tasks in a job).

The Lastword language gives Starfish another unique advantage. Note that Starfish is primarily a system for running analytics work-

[1] *La*nguage for *St*arfish *Wor*kloads and *D*ata.

[2] Cycles may be needed to support loops or iterative computations.

	WordCount		TeraSort	
	Rules of Thumb	Based on Job Profile	Rules of Thumb	Based on Job Profile
io.sort.spill.percent	0.80	0.80	0.80	0.80
io.sort.record.percent	0.50	0.05	0.15	0.15
io.sort.mb	200	50	200	200
io.sort.factor	10	10	10	100
mapred.reduce.tasks	27	2	27	400
Running Time (sec)	785	407	891	606

Table 1: Parameter settings from rules of thumb and recommendations from job profiles for WordCount and TeraSort

loads on big data. At the same time, we want Starfish to be usable in environments where workloads are run directly on Hadoop without going through Starfish. Lastword enables Starfish to be used as a recommendation engine in these environments. The full or partial Hadoop workload from such an environment can be expressed in Lastword—we will provide tools to automate this step—and then input to Starfish which is run in a special *recommendation mode*. In this mode, Starfish uses its tuning features to recommend good configurations at the job, workflow, and workload levels; instead of running the workload with these configurations as Starfish would do in its normal usage mode.

3. JUST-IN-TIME JOB OPTIMIZATION

The response surfaces in Figure 4 show the impact of various job configuration parameter settings on the running time of two MapReduce programs in Hadoop. We use WordCount and TeraSort which are simple, yet very representative, MapReduce programs. The default experimental setup used in this paper is a single-rack Hadoop cluster running on 16 Amazon EC2 nodes of the c1.medium type. Each node runs at most 3 map tasks and 2 reduce tasks concurrently. WordCount processes 30GB of data generated using the RandomTextWriter program in Hadoop. TeraSort processes 50GB of data generated using Hadoop's TeraGen program.

Rules of Thumb for Parameter Tuning: The job configuration parameters varied in Figure 4 are *io.sort.mb*, *io.sort.record.percent*, and *mapred.reduce.tasks*. All other parameters are kept constant. Table 1 shows the settings of various parameters for the two jobs based on popular rules of thumb used today [5, 13]. For example, the rules of thumb recommend setting *mapred.reduce.tasks* (the number of reduce tasks in the job) to roughly 0.9 times the total number of reduce slots in the cluster. The rationale is to ensure that all reduce tasks run in one wave while leaving some slots free for reexecuting failed or slow tasks. A more complex rule of thumb sets *io.sort.record.percent* to $\frac{16}{16+avg\_record\_size}$ based on the average size of map output records. The rationale here involves source-code details of Hadoop.

Figure 4 shows that the rule-of-thumb settings gave poor performance. In fact, the rule-of-thumb settings for WordCount gave one of its worst execution times: *io.sort.mb* and *io.sort.record.percent* were set too high. The interaction between these two parameters was very different and more complex for TeraSort as shown in Figure 4(b). A higher setting for *io.sort.mb* leads to better performance for certain settings of the *io.sort.record.percent* parameter, but hurts performance for other settings. The complexity of the surfaces and the failure of rules of thumb highlight the challenges a user faces if asked to tune the parameters herself. Starfish's job-level tuning components—Profiler, Sampler, What-if Engine, and Just-in-Time Optimizer—help automate this process.

Profiling Using Dynamic Instrumentation: The Profiler uses *dynamic instrumentation* to collect run-time monitoring information from unmodified MapReduce programs running on Hadoop. Dynamic instrumentation has become hugely popular over the last few years to understand, debug, and optimize complex systems [4]. The dynamic nature means that there is zero overhead when instrumentation is turned off; an appealing property in production deployments. The current implementation of the Profiler uses BTrace [2], a safe and dynamic tracing tool for the Java platform.

When Hadoop runs a MapReduce job, the Starfish Profiler dynamically instruments selected Java classes in Hadoop to construct a *job profile*. A profile is a concise representation of the job execution that captures information both at the task and subtask levels. The execution of a MapReduce job is broken down into the *Map Phase* and the *Reduce Phase*. Subsequently, the Map Phase is divided into the *Reading*, *Map Processing*, *Spilling*, and *Merging* subphases. The Reduce Phase is divided into the *Shuffling*, *Sorting*, *Reduce Processing*, and *Writing* subphases. Each subphase represents an important part of the job's overall execution in Hadoop.

The job profile exposes three views that capture various aspects of the job's execution:

1. *Timings view:* This view gives the breakdown of how wall-clock time was spent in the various subphases. For example, a map task spends time reading input data, running the user-defined map function, and sorting, spilling, and merging map-output data.

2. *Data-flow view:* This view gives the amount of data processed in terms of bytes and number of records during the various subphases.

3. *Resource-level view:* This view captures the usage trends of CPU, memory, I/O, and network resources during the various subphases of the job's execution. Usage of CPU, I/O, and network resources are captured respectively in terms of the time spent using these resources per byte and per record processed. Memory usage is captured in terms of the memory used by tasks as they run in Hadoop.

We will illustrate the benefits of job profiles and the insights gained from them through a real example. Figure 5 shows the Timings view from the profiles collected for the two configuration parameter settings for WordCount shown in Table 1. We will denote the execution of WordCount using the "Rules of Thumb" settings from Table 1 as Job A; and the execution of WordCount using the "Based on Job Profile" settings as Job B. Note that the same WordCount MapReduce program processing the same input dataset is being run in either case. The WordCount program uses a *Combiner* to perform reduce-style aggregation on the map task side for each spill of the map task's output. Table 1 shows that Job B runs 2x faster than Job A.

Our first observation from Figure 5 is that the map tasks in Job B completed on average much faster compared to the map tasks in Job A; yet the reverse happened to the reduce tasks. Further exploration of the Data-flow and Resource views showed that the Combiner in Job A was processing an extremely large number of records, causing high CPU contention. Hence, all the CPU-intensive operations in Job A's map tasks (executing the user-provided map function, serializing and sorting the map output) were negatively affected. Compared to Job A, the lower settings for *io.sort.mb* and *io.sort.record.percent* in Job B led to more, but individually smaller, map-side spills. Because the Combiner is invoked on these individually smaller map-side spills in Job B, the Combiner caused far less CPU contention in Job B compared to Job A.

On the other hand, the Combiner drastically decreases the amount of intermediate data that is spilled to disk as well as transferred over the network (*shuffled*) from map to reduce tasks. Since the map

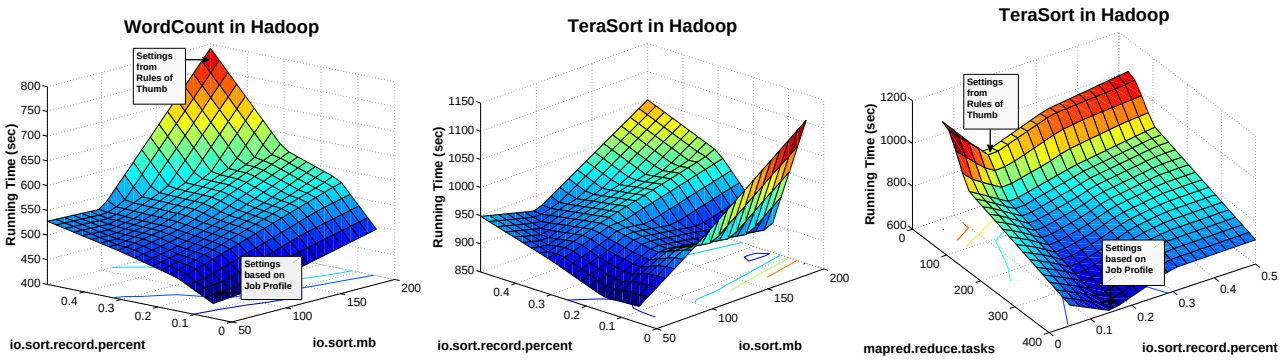


Figure 4: Response surfaces of MapReduce programs in Hadoop: (a) WordCount, with io.sort.mb $\in$ $[50, 200]$ and io.sort.record.percent $\in$ $[0.05, 0.5]$ (b) TeraSort, with io.sort.mb $\in$ $[50, 200]$ and io.sort.record.percent $\in$ $[0.05, 0.5]$ (c) TeraSort, with io.sort.record.percent $\in$ $[0.05, 0.5]$ and mapred.reduce.tasks $\in$ $[27, 400]$

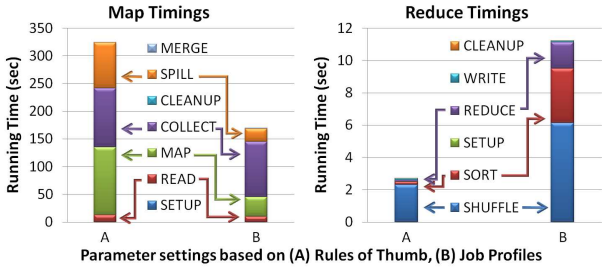


Figure 5: Map and reduce time breakdown for two WordCount jobs run with different settings of job configuration parameters

tasks in Job B processed smaller spills, the data reduction gains from the Combiner were also smaller; leading to larger amounts of data being shuffled and processed by the reducers. However, the additional local I/O and network transfer costs in Job B were dwarfed by the reduction in CPU costs.

Effectively, the more balanced usage of CPU, I/O, and network resources in the map tasks of Job B improved the overall performance of the map tasks significantly compared to Job A. Overall, the benefit gained by the map tasks in Job B outweighed by far the loss incurred by the reduce tasks; leading to the 2x better performance of Job B compared to the performance of Job A.

Predicting Job Performance in Hadoop: The job profile helps in understanding the job behavior as well as in diagnosing bottlenecks during job execution for the parameter settings used. More importantly, given a new setting S of the configuration parameters, the What-if Engine can use the job profile and a set of models that we developed to estimate the new profile if the job were to be run using S. This what-if capability is utilized by the Just-in-Time Optimizer in order to recommend good parameter settings.

The What-if Engine is given four inputs when asked to predict the performance of a MapReduce job J:

1. The job profile generated for J by the Profiler. The profile may be available from a previous execution of J. Otherwise, the Profiler can work in conjunction with Starfish's Sampler to generate an approximate job profile efficiently. Figure 6 considers approximate job profiles later in this section.

2. The new setting S of the job configuration parameters using which Job J will be run.

3. The size, layout, and compression information of the input dataset on which Job J will be run. Note that this input dataset can be different from the dataset used while generating the job profile.

4. The cluster setup and resource allocation that will be used to run Job J. This information includes the number of nodes and network topology of the cluster, the number of map and reduce task slots per node, and the memory available for each task execution.

The What-if Engine uses a set of *performance models* for predicting (a) the flow of data going through each subphase in the job's execution, and (b) the time spent in each subphase. The What-if Engine then produces a *virtual job profile* by combining the predicted information in accordance with the cluster setup and resource allocation that will be used to run the job. The virtual job profile contains the predicted Timings and Data-flow views of the job when run with the new parameter settings. The purpose of the virtual profile is to provide the user with more insights on how the job will behave when using the new parameter settings, as well as to expand the use of the What-if Engine towards answering hypothetical questions at the workflow and workload levels.

Towards Cost-Based Optimization: Table 1 shows the parameter settings for WordCount and TeraSort recommended by an initial implementation of the Just-in-Time Optimizer. The What-if Engine used the respective job profiles collected from running the jobs using the rules-of-thumb settings. WordCount runs almost twice as fast at the recommended setting. As we saw earlier, while the Combiner reduced the amount of intermediate data drastically, it was making the map execution heavily CPU-bound and slow. The configuration setting recommended by the optimizer—with lower *io.sort.mb* and *io.sort.record.percent*—made the map tasks significantly faster. This speedup outweighed the lowered effectiveness of the Combiner that caused more intermediate data to be shuffled and processed by the reduce tasks.

These experiments illustrate the usefulness of the Just-in-Time Optimizer. One of the main challenges that we are addressing is in developing an efficient strategy to search through the high-dimensional space of parameter settings. A related challenge is in generating job profiles with minimal overhead. Figure 6 shows the tradeoff between the profiling overhead (in terms of job slowdown) and the average relative error in the job profile views when profiling is limited to a fraction of the tasks in WordCount. The results are promising but show room for improvement.

4. WORKFLOW-AWARE SCHEDULING

Cause and Effect of Unbalanced Data Layouts: Section 2.2 mentioned how interactions between the task scheduler and the policies employed by the distributed filesystem can lead to unbalanced data layouts. Figure 7 shows how even the execution of a single large

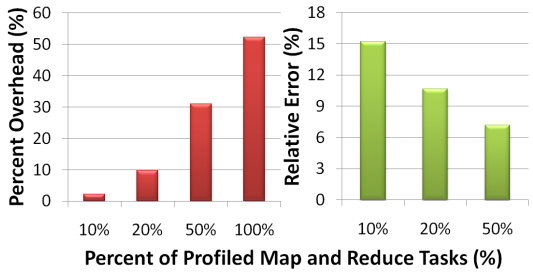


Figure 6: (a) Relative job slowdown, and (b) relative error in the approximate views generated as the percentage of profiled tasks in a job is varied

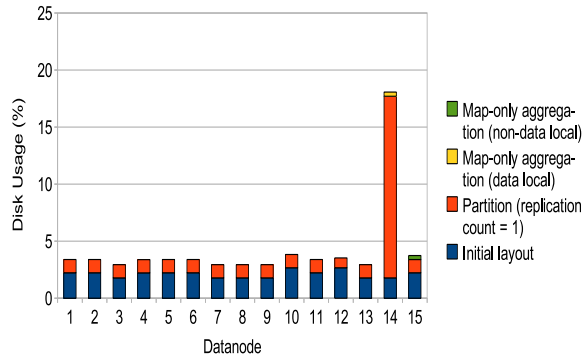


Figure 7: Unbalanced data layout

job can cause an unbalanced layout in Hadoop. We ran a partitioning MapReduce job (similar to "Partition by age" shown in Figure 2) that partitions a 100GB TPC-H Lineitem table into four partitions relevant to downstream workflow nodes. The data properties are such that one partition is much larger than the others. All the partitions are replicated once as done by default for intermediate workflow data in systems like Pig [11]. HDFS ends up placing all blocks for the large partition on the node (Datanode 14) where the reduce task generating this partition runs.

A number of other causes can lead to unbalanced data layouts rapidly or over time: (a) skewed data, (b) scheduling of tasks in a data-layout-unaware manner as done by the Hadoop schedulers available today, and (c) addition or dropping of nodes without running costly data rebalancing operations. (HDFS does not automatically move existing data when new nodes are added.) Unbalanced data layouts are a serious problem in big data analytics because they are prominent causes of task failure (due to insufficient free disk space for intermediate map outputs or reduce inputs) and performance degradation. We observed a more than 2x slowdown for a sort job running on the unbalanced layout in Figure 7 compared to a balanced layout.

Unbalanced data layouts cause a dilemma for data-locality-aware schedulers (i.e., schedulers that aim to move computation to the data). Exploiting data locality can have two undesirable consequences in this context: performance degradation due to reduced parallelism, and worse, making the data layout further unbalanced because new outputs will go to the over-utilized nodes. Figure 7 also shows how running a map-only aggregation on the large partition leads to the aggregation output being written to the over-utilized Datanode 14. The aggregation output was small. A larger output could have made the imbalance much worse. On the other hand, non-data-local scheduling (i.e., moving data to the computation) incurs the overhead of data movement. A useful new feature in Hadoop will be to piggyback on such data movements to rebalance the data layout.

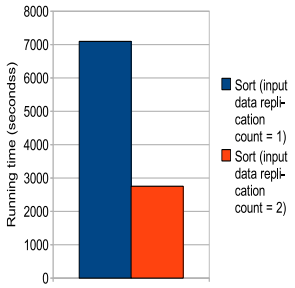


Figure 8: Sort running time on the partitions

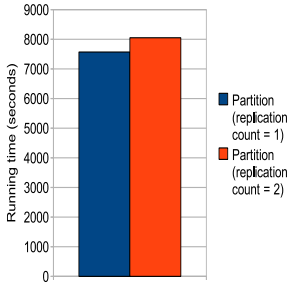


Figure 9: Partition creation time

We ran the same partitioning job with a replication factor of two for the partitions. For our single-rack cluster, HDFS places the second replica of each block of the partitions on a randomly-chosen node. The overall layout is still unbalanced, but the time to sort the partitions improved significantly because the second copy of the data is spread out over the cluster (Figure 8). Interestingly, as shown in Figure 9, the overhead of creating a second replica is very small on our cluster (which will change if the network becomes the bottleneck [11]).

Aside from ensuring that the data layout is balanced, other choices are available such as collocating two or more datasets. Consider a workflow consisting of three jobs. The first two jobs partition two separate datasets R and S (e.g., Users and GeoInfo from Figure 2) using the same partitioning function into n partitions each. The third job, whose input consists of the outputs of the first two jobs, performs an equi-join of the respective partitions from R and S. HDFS does not provide the ability to collocate the joining partitions from R and S; so a join job run in Hadoop will have to do non-data-local reads for one of its inputs.

We implemented a new block placement policy in HDFS that enables collocation of two or more datasets. (As an excellent example of Hadoop's extensibility, HDFS provides a pluggable interface that simplifies the task of implementing new block placement policies [9].) Figure 10 shows how the new policy gives a 22% improvement in the running time of a partition-wise join job by collocating the joining partitions.

Experimental results like those above motivate the need for a Workflow-aware Scheduler that can run jobs in a workflow such that the overall performance of the workflow is optimized. Workflow performance can be measured in terms of running time, resource utilization in the Hadoop cluster, and robustness to failures (e.g., minimizing the need for cascading reexecution of tasks due to node failure or data corruption) and transient issues (e.g., reacting to the slowdown of a node due to temporary resource contention). As illustrated by Figures 7–10, good layouts of the initial (base), intermediate (temporary), and final (results) data in a workflow are vital to ensure good workflow performance.

Workflow-aware Scheduling: A Workflow-aware Scheduler can ensure that job-level optimization and scheduling policies are co-

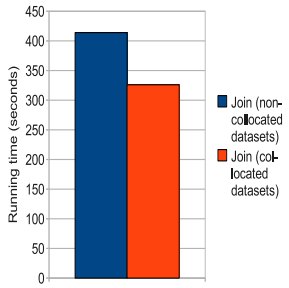


Figure 10: Respective execution times of a partition-wise join job with noncollocated and collocated input partitions

ordinated tightly with the policies for data placement employed by the underlying distributed filesystem. Rather than making decisions that are locally optimal for individual MapReduce jobs, Starfish's Workflow-aware Scheduler makes decisions by considering producer-consumer relationships among jobs in the workflow.

Figure 11 gives an example of producer-consumer relationships among three Jobs P, $C1$, and $C2$ in a workflow. Analyzing these relationships gives important information such as:

- What parts of the data output by a job are used by downstream jobs in the workflow? Notice from Figure 11 that the three writer tasks of Job P generate files *File1*, *File2*, and *File3* respectively. (In a MapReduce job, the writer tasks are map tasks in a map-only job, and reduce tasks otherwise.) Each file is stored as blocks in the distributed filesystem. (HDFS blocks are 64MB in size by default.) *File1* forms the input to Job $C1$, while *File1* and *File2* form the input to Job $C2$. Since *File3* is not used by any of the downstream jobs, a Workflow-aware Scheduler can configure Job P to avoid generating *File3*.
- What is the unit of data-level parallelism in each job that reads the data output by a job? Notice from Figure 11 that the data-parallel reader tasks of Job $C1$ read and process one data block each. However, the data-parallel reader tasks of Job $C2$ read one file each. (In a MapReduce job in a workflow, the data-parallel map tasks of the job read the output of upstream jobs in the workflow.) While not shown in Figure 11, jobs like the join in Figure 10 consist of data-parallel tasks that each read a group of files output by upstream jobs in the workflow. Information about the data-parallel access patterns of jobs is vital to guarantee good data layouts that, in turn, will guarantee an efficient mix of parallel and data-local computation. For *File2* in Figure 11, all blocks in the file should be placed on the same node to ensure data-local computation (i.e., to avoid having to move data to the computation). The choice for *File1*, which is read by both Jobs $C1$ and $C2$, is not so easy to make. The data-level parallelism is at the block-level in Job $C1$, but at the file-level in Job $C2$. Thus, the optimal layout of *File1* from Job $C1$'s perspective is to spread *File1*'s blocks across the nodes so that $C1$'s map tasks can run in parallel across the cluster. However, the optimal layout of *File1* from Job $C2$'s perspective is to place all blocks on the same node.

Starfish's Workflow-aware Scheduler works in conjunction with the What-if Engine and the Just-in-Time Optimizer in order to pick the job execution schedule as well as the data layouts for a workflow. The space of choices for data layout includes:

1. What block placement policy to use in the distributed filesystem for the output file of a job? HDFS uses the *Local Write* block placement policy which works as follows: the first replica of any block is stored on the same node where the block's writer (a map or reduce task) runs. We have implemented a new *Round Robin* block placement policy in HDFS where the blocks written are stored on the nodes of the distributed filesystem in a round robin fashion.
2. How many replicas to store—called the *replication factor*—for the blocks of a file? Replication helps improve performance for heavily-accessed files. Replication also improves robustness by reducing performance variability in case of node failures.
3. What size to use for blocks of a file? For a very big file, a block size larger than the default of 64MB can improve performance significantly by reducing the number of map tasks needed to process the file. The caveat is that the choice of the block size interacts with the choice of job-level configuration parameters like *io.sort.mb* (recall Section 3).
4. Should a job's output files be compressed for storage? Like the use of Combiners (recall Section 3), the use of compression enables the cost of local I/O and network transfers to be traded for additional CPU cost. Compression is not always beneficial. Furthermore, like the choice of the block size, the usefulness of compression depends on the choice of job-level parameters.

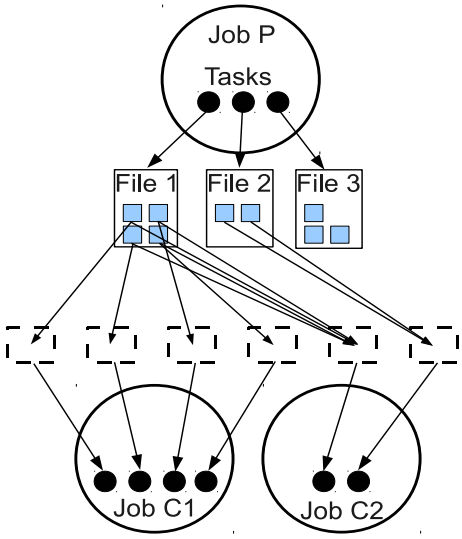


Figure 11: Part of an example workflow showing producer-consumer relationships among jobs

The Workflow-aware Scheduler performs a cost-based search for a good layout for the output data of each job in a given workflow. The technique we employ here asks a number of questions to the What-if Engine; and uses the answers to infer the costs and benefits of various choices. The what-if questions asked for a workflow consisting of the producer-consumer relationships among Jobs P, $C1$, and $C2$ shown in Figure 11 include:

(a) What is the expected running time of Job P if the Round Robin block placement policy is used for P's output files?

(b) What will the new data layout in the cluster be if the Round Robin block placement policy is used for P's output files?

(c) What is the expected running time of Job $C1$ ($C2$) if its input data layout is the one in the answer to Question (b)?

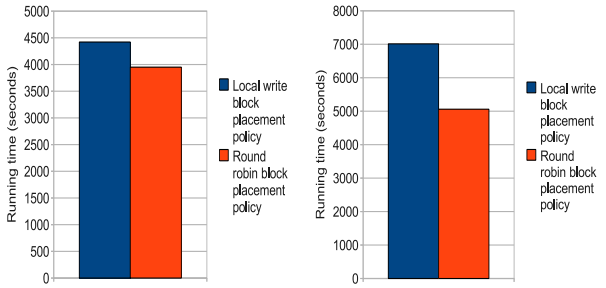


Figure 12: Respective running times of (a) a partition job and (b) a two-job workflow with the (default) Local Write and the Round Robin block placement policies used in HDFS

(d) What are the expected running times of Jobs $C1$ and $C2$ if they are scheduled concurrently when Job P completes?

(e) Given the Local Write block placement policy and a replication factor of 1 for Job P's output, what is the expected increase in the running time of Job $C1$ if one node in the cluster were to fail during $C1$'s execution?

These questions are answered by the What-if Engine based on a simulation of the main aspects of workflow execution. This step involves simulating MapReduce job execution, task scheduling, and HDFS block placement policies. The job-level and cluster-level information described in Section 3 is needed as input for the simulation of workflow execution.

Figure 12 shows results from an experiment where the Workflow-aware Scheduler was asked to pick the data layout for a two-job workflow consisting of a partition job followed by a sort job. The choice for the data layout involved selecting which block placement policy to use between the (default) Local Write policy and the Round Robin policy. The remaining choices were kept constant: replication factor is 1, the block size is 128MB, and compression is not used. The choice of collocation was not considered since it is not beneficial to collocate any group of datasets in this case.

The Workflow-aware Scheduler first asks what-if questions regarding the partition job. The What-if Engine predicted correctly that the Round Robin policy will perform better than the Local Write policy for the output data of the partition job. In our cluster setting on Amazon EC2, the local I/O within a node becomes the bottleneck before the parallel writes of data blocks to other storage nodes over the network. Figure 12(a) shows the actual performance of the partition job for the two block placement policies.

The next set of what-if questions have to do with the performance of the sort job for different layouts of the output of the partition job. Here, using the Round Robin policy for the partition job's output emerges a clear winner. The reason is that the Round Robin policy spreads the blocks over the cluster so that maximum data-level parallelism of sort processing can be achieved while performing data-local computation. Overall, the Workflow-aware Scheduler picks the Round Robin block placement policy for the entire workflow. As seen in Figure 12(b), this choice leads to the minimum total running time of the two-job workflow. Use of the Round Robin policy gives around 30% reduction in total running time compared to the default Local Write policy.

5. OPTIMIZATION AND PROVISIONING FOR HADOOP WORKLOADS

Workload Optimizer: Starfish's Workload Optimizer represents the workload as a directed graph and applies the optimizations listed in Section 2.3 as graph-to-graph transformations. The optimizer

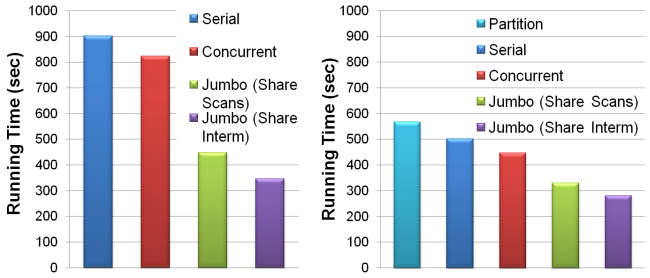


Figure 13: Processing multiple SPA workflow nodes on the same input dataset

uses the What-if Engine to do a cost-based estimation of whether the transformation will improve performance.

Consider the workflows that produce the results IV, V, and VI in Figure 2. These workflows have a join of Users and Clicks in common. The results IV, V, and VI can each be represented as a Select-Project-Aggregate (SPA) expression over the join. Starfish has an operator, called the *Jumbo operator*, that can process any number of logical SPA workflow nodes over the same table in a single MapReduce job. (MRShare [14] and Pig [15] also support similar operators.) Without the Jumbo operator, each SPA node will have to be processed as a separate job. The Jumbo operator enables sharing of all or some of the map-side scan and computation, sorting and shuffling, as well as the reduce-side scan, computation, and output generation. At the same time, the Jumbo operator can help the scheduler to better utilize the bounded number of map and reduce task slots in a Hadoop cluster.

Figure 13(a) shows an experimental result where three logical SPA workflow nodes are processed on a 24GB dataset as: (a) *Serial*, which runs three separate MapReduce jobs in sequence; (b) *Concurrent*, which runs three separate MapReduce jobs concurrently; (c) using the Jumbo operator to share the map-side scans in the SPA nodes; and (d) using the Jumbo operator to share the map-side scans as well as the intermediate data produced by the SPA nodes. Figure 13(a) shows that sharing the sorting and shuffling of intermediate data, in addition to sharing scans, provides additional performance benefits.

Now consider the workflows that produce results I, II, IV, and V in Figure 2. These four workflows have filter conditions on the age attribute in the Users dataset. Running a MapReduce job to partition Users based on ranges of age values will enable the four workflows to prune out irrelevant partitions efficiently. Figure 13(b) shows the results from applying partition pruning to the same three SPA nodes from Figure 13(a). Generating the partitions has significant overhead—as seen in Figure 13(b)—but possibilities exist to hide or reduce this overhead by combining partitioning with a previous job like data copying. Partition pruning improves the performance of all MapReduce jobs in our experiment. At the same time, partition pruning decreases the performance benefits provided by the Jumbo operator. These simple experiments illustrate the interactions among different optimization opportunities that exist for Hadoop workloads.

Elastisizer: Users can now leverage pay-as-you-go resources on the cloud to meet their analytics needs. Amazon Elastic MapReduce allows users to instantiate a Hadoop cluster on EC2 nodes, and run workflows. The typical workflow on Elastic MapReduce accesses data initially from S3, does in-cluster analytics, and writes final output back to S3 (Figure 2). The cluster can be released when the workflow completes, and the user pays for the resources used. While Elastic MapReduce frees users from setting up and maintaining Hadoop clusters, the burden of cluster provisioning is still

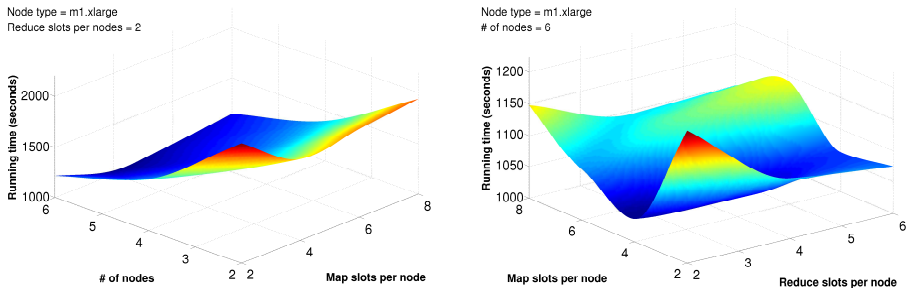


Figure 14: Workload performance under various cluster and Hadoop configurations on Amazon Elastic MapReduce

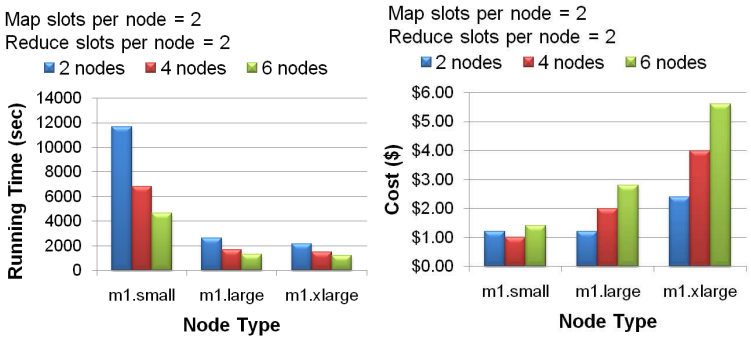


Figure 15: Performance Vs. pay-as-you-go costs for a workload on Amazon Elastic MapReduce

on the user. Specifically, users have to specify the number and type of EC2 nodes (from among 10+ types) as well as whether to copy data from S3 into the in-cluster HDFS. The space of provisioning choices is further complicated by Amazon Spot Instances which provide a market-based option for leasing EC2 nodes. In addition, the user has to specify the Hadoop-level as well as job-level configuration parameters for the provisioned cluster.

One of the goals of Starfish's Elastisizer is to automatically determine the best cluster and Hadoop configurations to process a given workload subject to user-specified goals (e.g., on completion time and monetary costs incurred). To illustrate this problem, Figure 14 shows how the performance of a workload W consisting of a single workflow varies across different cluster configurations (number and type of EC2 nodes) and corresponding Hadoop configurations (number of concurrent map and reduce slots per node).

The user could have multiple preferences and constraints for the workload, which poses a multi-objective optimization problem. For example, the goal may be to minimize the monetary cost incurred to run the workload, subject to a maximum tolerable workload completion time. Figures 15(a) and 15(b) show the running time as well as cost incurred on Elastic MapReduce for the workload W for different cluster configurations. Some observations from the figures:

- If the user wants to minimize costs subject to a completion time of 30 minutes, then the Elastisizer should recommend a cluster of four m1.large EC2 nodes.
- If the user wants to minimize costs, then two m1.small nodes are best. However, the Elastisizer can suggest that by paying just 20% more, the completion time can be reduced by 2.6x.

To estimate workload performance for various cluster configurations, the Elastisizer invokes the What-if Engine which, in turn, uses a mix of simulation and model-based estimation. As discussed in Section 4, the What-if Engine simulates the task scheduling and block-placement policies over a hypothetical cluster, and uses performance models to predict the data flow and performance of the MapReduce jobs in the workload. The latest Hadoop release includes a Hadoop simulator, called *Mumak*, that we initially attempted to use in the What-if Engine. However, Mumak needs a workload execution trace for a specific cluster size as input, and cannot simulate workload execution for a different cluster size.

6. RELATED WORK AND SUMMARY

Hadoop is now a viable competitor to existing systems for big data analytics. While Hadoop currently trails existing systems in peak query performance, a number of research efforts are addressing this issue [1, 7, 10]. Starfish fills a different void by enabling Hadoop users and applications to get good performance automatically throughout the data lifecycle in analytics; without any need on their part to understand and manipulate the many tuning knobs available. A system like Starfish is essential as Hadoop usage continues to grow beyond companies like Facebook and Yahoo! that have considerable expertise in Hadoop. New practitioners of big data analytics like computational scientists and systems researchers lack the expertise to tune Hadoop to get good performance.

Starfish's tuning goals and solutions are related to projects like Hive, Manimal, MRShare, Nectar, Pig, Quincy, and Scope [3, 8, 14, 15, 18]. The novelty in Starfish's approach comes from how it focuses *simultaneously* on different workload granularities—overall workload, workflows, and jobs (procedural and declarative)—as well as across various decision points—provisioning, optimization, scheduling, and data layout. This approach enables Starfish to handle the significant interactions arising among choices made at different levels.

7. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[2] *BTrace: A Dynamic Instrumentation Tool for Java*. `http://kenai.com/projects/btrace`.

[3] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.

[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference*, 2004.

[5] Cloudera: 7 tips for Improving MapReduce Performance. `http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance`.

[6] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[7] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1), 2010.

[8] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, 2010.

[9] Pluggable Block Placement Policies in HDFS. `issues.apache.org/jira/browse/HDFS-385`.

[10] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1), 2010.

[11] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making Cloud Intermediate Data Fault-tolerant. In *SoCC*, 2010.

[12] TeraByte-scale Data Cycle at LinkedIn. `http://tinyurl.com/lukod6`.

[13] Hadoop MapReduce Tutorial. `http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html`.

[14] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1), 2010.

[15] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX Annual Technical Conference*, 2008.

[16] A. Qureshi, R. Weber, H. Balakrishnan, J. V. Guttag, and B. V. Maggs. Cutting the Electric Bill for Internet-scale Systems. In *SIGCOMM*, 2009.

[17] Agile Enterprise Analytics. Keynote by Oliver Ratzesberger at the Self-Managing Database Systems Workshop 2010.

[18] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, 2010.

APPENDIX

A. STARFISH'S VISUALIZER

When a MapReduce job executes in a Hadoop cluster, a lot of information is generated including logs, counters, resource utilization metrics, and profiling data. This information is organized, stored, and managed by Starfish's Metadata Manager in a catalog that can be viewed using Starfish's *Visualizer*. A user can employ the Visualizer to get a deep understanding of a job's behavior during execution, and to ultimately tune the job. Broadly, the functionality of the Visualizer can be categorized into *Timeline views*, *Data-flow views*, and *Profile views*.

A.1 Timeline Views

Timeline views are used to visualize the progress of a job execution at the task level. Figure 16 shows the execution timeline of map and reduce tasks that ran during a MapReduce job execution. The user can observe information like how many tasks were running at any point in time on each node, when each task started and ended, or how many map or reduce waves occurred. The user is able to quickly spot any variance in the task execution times and discover potential load balancing issues.

Moreover, Timeline views can be used to compare different executions of the same job run at different times or with different parameter settings. Comparison of timelines will show whether the job behavior changed over time as well as help understand the impact that changing parameter settings has on job execution. In addition, the Timeline views support a *What-if* mode using which the user can visualize what the execution of a job will be when run using different parameter settings. For example, the user can determine the impact of decreasing the value of *io.sort.mb* on map task execution. Under the hood, the Visualizer invokes the What-if Engine to generate a virtual job profile for the job in the hypothetical setting (recall Section 3).

A.2 Data-flow Views

The Data-flow views enable visualization of the flow of data among the nodes and racks of a Hadoop cluster, and between the map and reduce tasks of a job. They form an excellent way of identifying data skew issues and realizing the need for a better partitioner in a MapReduce job. Figure 17 presents the data flow among the nodes during the execution of a MapReduce job. The thickness of each line is proportional to the amount of data that was shuffled between the corresponding nodes. The user also has the ability to specify a set of filter conditions (see the left side of Figure 17) that allows her to zoom in on a subset of nodes or on the large data transfers. An important feature of the Visualizer is the *Video mode* that allows users to play back a job execution from the past. Using the Video mode (Figure 17), the user can inspect how data was processed and transfered between the map and reduce tasks of the job, and among nodes and racks of the cluster, as time went by.

A.3 Profile Views

In Section 3, we saw how a job profile contains a lot of useful information like the breakdown of task execution timings, resource usage, and data flow per subphase. The Profile views help visualize the job profiles, namely, the information exposed by the Timings, Data-flow, and Resource-level views in a profile; allowing an in-depth analysis of the task behavior during execution. For example, Figure 5 shows parts of two Profile views that display the breakdown of time spent on average in each map and reduce task for two WordCount job executions. Job A was run using the parameter settings as specified by rules of thumb, whereas Job B was run using the settings recommended by the Just-in-time Optimizer (Table 1 in Section 3). The main difference caused by the two settings was more, but smaller, map-side spills for Job B compared to Job A.

We can observe that the map tasks in Job B completed on average much faster compared to the map tasks in Job A; yet the reverse happened to the reduce tasks. The Profile views allow us to see exactly which subphases benefit the most from the parameter settings. It is obvious from Figure 5 that the time spent performing the map processing and the spilling in Job B was significantly lower compared to Job A.

On the other hand, the Combiner drastically decreases the amount of intermediate data spilled to disk (which can be observed in the Data-flow views not shown here). Since the map tasks in Job B processed smaller spills, the reduction gains from the Combiner were also smaller; leading to larger amounts of data being shuffled and processed by the reducers. The Profile views show exactly how much more time was spent in Job B for shuffling and sorting the intermediate data, as well as performing the reduce computation. Overall, the benefit gained by the map tasks in Job B outweighed by far the loss incurred by the reduce tasks, leading to a 2x better performance than Job A.

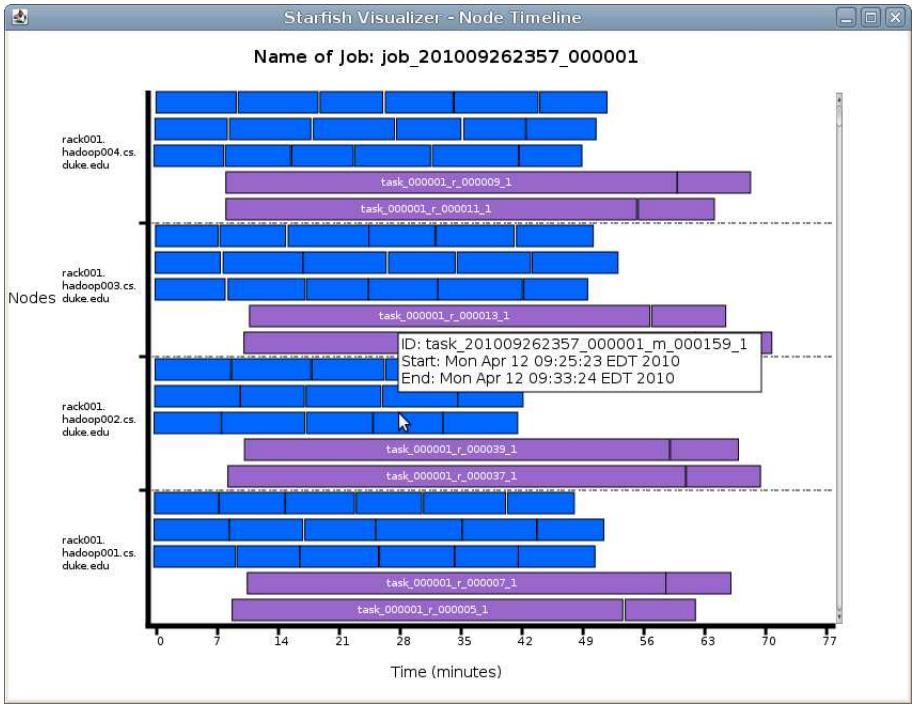


Figure 16: Execution timeline of the map and reduce tasks of a MapReduce job

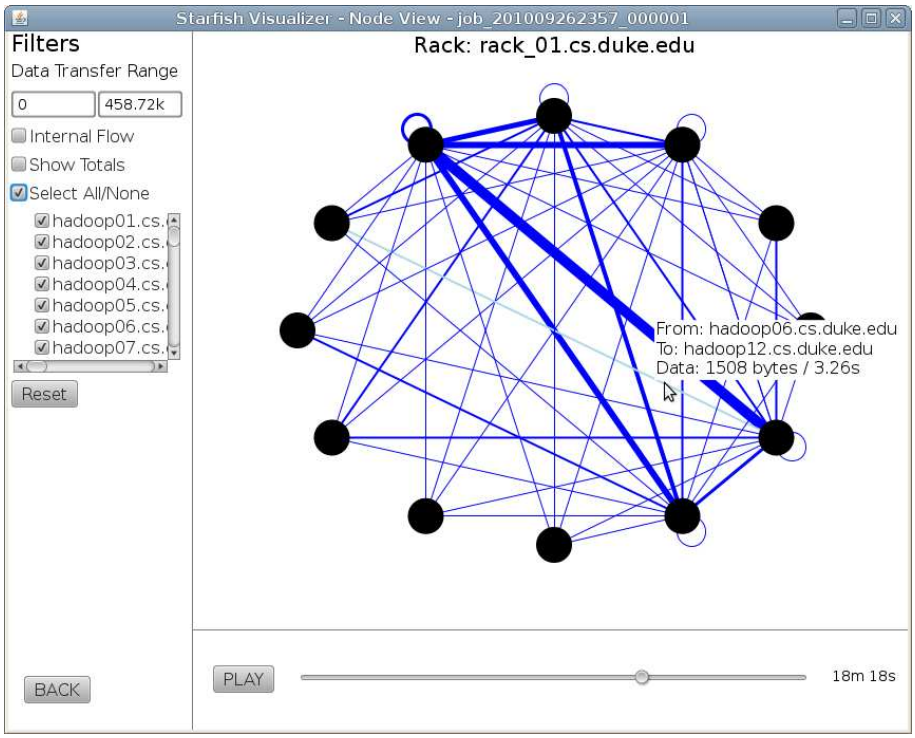


Figure 17: Visual representation of the data-flow among the Hadoop nodes during a MapReduce job execution

Provenance for Generalized Map and Reduce Workflows*

Robert Ikeda, Hyunjung Park, and Jennifer Widom
Stanford University
{rmikeda,hyunjung,widom}@cs.stanford.edu

ABSTRACT

We consider a class of workflows, which we call *generalized map and reduce workflows* (*GMRWs*), where input data sets are processed by an acyclic graph of *map* and *reduce* functions to produce output results. We show how *data provenance* (also sometimes called *lineage*) can be captured for map and reduce functions transparently. The captured provenance can then be used to support *backward tracing* (finding the input subsets that contributed to a given output element) and *forward tracing* (determining which output elements were derived from a particular input element). We provide formal underpinnings for provenance in GMRWs, and we identify properties that are guaranteed to hold when provenance is applied recursively. We have built a prototype system that supports provenance capture and tracing as an extension to Hadoop. Our system uses a wrapper-based approach, requiring little if any user intervention in most cases, and retaining Hadoop's parallel execution and fault tolerance. Performance numbers from our system are reported.

1. INTRODUCTION

Data-oriented workflows are graphs where nodes denote dataset *transformations*, and edges denote the flow of data input to and output from the transformations. Such workflows are common in, e.g., scientific data processing [8, 14, 23] and information extraction [22]. A special case of such workflows is what we refer to as *generalized map and reduce workflows* (*GMRWs*), in which all transformations are either *map* or *reduce* functions [3, 15]. Our setting is more general than conventional MapReduce jobs, which have just one map function followed by one reduce function; rather we consider any acyclic graph of map and reduce functions.

In data-oriented workflows, it can be useful to track *data provenance* (also sometimes called *lineage*), capturing how data elements are processed through the workflow [1, 8, 11, 23]. Provenance supports *backward tracing* (finding the input subsets that contributed to a given output element) and *forward tracing* (determining which output elements were derived from a particular input element). Backward tracing can be useful for, e.g., debugging and drilling-down, while forward tracing can be useful for, e.g., tracking error propagation. In addition, provenance can form the basis for *incremental maintenance* [16] and *selective refresh* [18].

In this paper, we explore data provenance for forward and backward tracing in GMRWs. In particular, we will see that the special case of workflows where all transformations are map or reduce functions allows us to define, capture, and exploit provenance more easily and efficiently than for general data-oriented workflows. We will also see that provenance can be captured for both map and reduce functions transparently using wrappers in Hadoop [3].

There has been a large body of work in provenance, including for general workflows (Section 1.1). Although map and reduce functions as data transformations have become increasingly popular, we are unaware of any work that focuses specifically on provenance for GMRWs. Provenance can be defined naturally for individual map and reduce functions, but it turns out to be a challenge to identify properties that hold when one-level provenance is applied recursively through a workflow. Also, we explore the overhead of provenance capture and the cost of provenance tracing. Our goal is to enable efficient provenance tracing in GMRWs while keeping the capture overhead low. Overall, our contributions are as follows:

- After establishing foundations in Section 2, in Section 3 we define provenance for individual map and reduce functions. We then identify properties that hold when one-level provenance is applied recursively through a GMRW.
- Section 4 describes how provenance can be captured and stored during workflow execution, and it specifies backward and forward tracing procedures using provenance.
- We have built a system called *RAMP* (*Reduce And Map Provenance*) that implements the concepts in this paper. Section 5 describes the implementation of RAMP as an extension to Hadoop. RAMP wraps Hadoop components automatically, requiring little if any user intervention, and retaining Hadoop's parallel execution and fault tolerance.
- Section 6 reports performance results using RAMP on the time and space overhead of capturing provenance, and discusses the cost of provenance tracing in our current system.

In Section 7, we conclude and discuss future work, including how we can incorporate SQL processing into GMRWs with provenance.

1.1 Related Work

Obviously there has been tremendous interest recently in high-performance parallel data processing specified via map and reduce functions, e.g., [3, 15, 27]. In addition, higher-level platforms have been built on top of these systems to make data-parallel programming easier, e.g., [9, 20, 24]. Regardless of which level they operate on, none of these systems or frameworks provides explicit functionality or even formal underpinnings for provenance.

At the same time, there has been a large body of work in lineage

*This work was supported by grants from the National Science Foundation (IIS-0904497) and Amazon Web Services.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

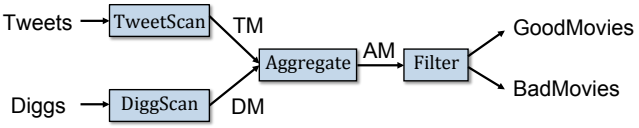


Figure 1: Movie sentiment workflow example.

and provenance over the past two decades, surveyed in, e.g., [8, 11, 23]. Provenance specifically in the data-oriented workflow setting is considered by [1, 2, 7, 10, 12, 17, 18, 19, 26], among others. However, none of this work considers the specific case of GMRWs, whose special properties and opportunities in the context of provenance are the focus of this paper.

Reference [12]—our own work from the distant past—is perhaps most related. It provides a hierarchy of transformation types relevant to provenance; each transformation is placed in the hierarchy by its creator to make provenance tracing as efficient as possible. Our map and reduce functions fall into the hierarchy, but they are specific enough that we can capture provenance automatically using a wrapper-based approach. Also, while [12] allows acyclic graphs of transformations, it does not investigate behavioral properties when provenance is traced recursively through them. We show in this paper that recursive provenance tracing can yield ill-behaved results in certain subtle cases. Finally, in this paper we consider the overhead incurred gathering extra information during workflow execution to facilitate provenance tracing, a topic not considered in [12].

1.2 Running Example

As a simplified example GMRW that serves primarily to illustrate our definitions and techniques, consider the workflow shown in Figure 1, used to gauge public opinion on movies. The inputs to the workflow are data sets Tweets and Diggs, containing user postings collected from Twitter and Digg, respectively. (Note we consider batch processing of data sets, not continuous stream processing.) The workflow involves the following transformations:

- Map functions **TweetScan** and **DiggScan** analyze the postings in data sets Tweets and Diggs, looking for postings that contain a single movie title and one or more positive or negative adjectives. For each such posting, a key-value pair is emitted to TwitterMovies (TM) or DiggMovies (DM), where the key is the title of the movie, and the value is a rating between 1 and 10 based on the combination of adjectives appearing.
- Reduce function **Aggregate** computes the number of ratings and the median rating for each movie title, producing data set AggMovies (AM).
- Map function **Filter** copies to GoodMovies those movies with at least 1000 ratings and a median rating of 6 or higher, and copies to BadMovies those movies with at least 1000 ratings and a median rating of 5 or lower.

As a simple example of how provenance might be useful in this workflow, suppose we are surprised to see that *Twilight* is in GoodMovies. Tracing provenance back one level to AggMovies, we see that *Twilight* has a median rating of 9, with over 1000 ratings. Further tracing provenance all the way back to the original postings, we sample usernames of *Twilight* fans. By reading other postings by these fans, we infer that teenage girls in particular have been flooding social media sites with raves for *Twilight*.

2. TRANSFORMATIONS & WORKFLOWS

Let a *data set* be any set of data *elements*. We assume every element has a unique identifier (discussed later). Thus, there are no duplicates in any data set. A *transformation* T is any procedure that takes one or more data sets as input and produces one or more data sets as output. A *workflow* is a directed acyclic graph, where nodes are transformations, and each edge is annotated with a data set.

In *generalized map and reduce workflows*, the two types of transformations are *map functions* and *reduce functions*. For now, we consider map and reduce functions with just one input set and one output set; Section 2.2 explains how multiple input and output sets are handled.

Map Functions. As in the MapReduce framework, a *map function* M produces zero or more output elements independently for each element in its input set I: $M(I) = \bigcup_{i \in I} M(\{i\})$. In practice, programmers in the MapReduce framework are not prevented from writing map functions that buffer the input or otherwise use "side-effect" temporary storage, resulting in behavior that violates this pure definition of a map function. In this paper, we assume pure map functions.

Reduce Functions. A *reduce function* R takes an input data set I in which each element is a key-value pair, and returns zero or more output elements independently for each group of elements in I with the same key: Let $k_1, \ldots, k_n$ be all of the distinct keys in I. Then $R(I) = \bigcup_{1 \le j \le n} R(G_j)$, where each G_j consists of all key-value pairs in I with key k_j. Similar to map functions, we consider only pure reduce functions, i.e., those satisfying this definition. In the remainder of the paper, we use $G_1, .., G_n$ to denote the key-based *groups* of a reduce function's input set I.

2.1 Transformation Properties

We now list some properties that are relevant for provenance.

Deterministic Functions. We assume that all functions are *deterministic*: Each map and reduce function returns the same output set when given the same input set. Again, programmers in the MapReduce framework are not prevented from creating nondeterministic functions, but we assume determinism in this paper.

Multiplicity for Map Functions. We say that a map function M is *one-one* if for any input set I, each element in I produces at most one output element: For all $i \in I$, $|M(\{i\})| \le 1$. Otherwise, the map function is *one-many*. In our running example, **TweetScan**, **DiggScan**, and **Filter** are all one-one.

Multiplicity for Reduce Functions. We say that a reduce function R is *many-one* if for any input set I, each key-based group G_j of I returns at most one output element: $|R(G_j)| \le 1$. Otherwise, the reduce function is *many-many*. In our running example, **Aggregate** is many-one.

Monotonicity. We say that a transformation T is *monotonic* if for any input sets I and I' with $I \subseteq I'$, then $T(I) \subseteq T(I')$. Note that map functions are always monotonic, but some reduce functions are nonmonotonic. In our running example, **Aggregate** is nonmonotonic. An example of a monotonic reduce function is one that simply returns the key for all groups above a certain size.

A thorough analysis of transformation properties in the context of provenance was developed in [12]. When we place the map and reduce functions we consider into the hierarchy of that paper, our provenance definitions (Section 3) are consistent with the definitions in [12].

2.2 Union and Split Transformations

So far we have assumed map and reduce functions have a single input data set and single output data set. In practice, functions in

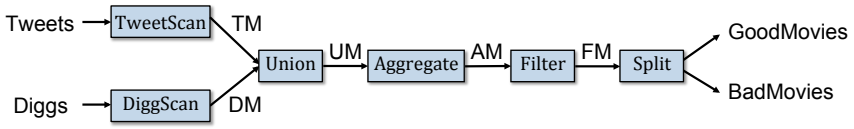


Figure 2: Movie workflow example with union and split.

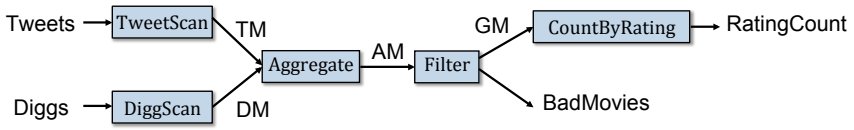


Figure 3: Modified movie workflow example with ill-behaved provenance.

the MapReduce framework can have multiple input data sets, but logically they union their input sets and then perform the function. Similarly, a map or reduce function with multiple output sets is logically equivalent to a function that outputs one large set, then splits it into multiple separate output sets. For our analysis in Section 3, it is preferable to model all map and reduce functions as single-input and single-output. Thus, we logically add union and split transformations to GMRWs, without changing their behavior.

A *union* transformation takes input data sets $I_1, \ldots, I_m$ and creates output set $O = I_1 \cup \ldots \cup I_m$. A *split* transformation takes input set I and creates output sets $O_1, .., O_r$, with $O_1 \cup \ldots \cup O_r = I$. For split, we assume that output sets are both deterministic and context-independent, i.e., each $i \in I$ is in the same O_k regardless of other elements in I.

Recall we assume all of our data sets have unique identifiers. We further assume identifiers are made globally unique, so $\cup$ in the above definitions is always disjoint union. Figure 2 adds union and split transformations to our running example.

3. PROVENANCE

Given a transformation instance $T(I) = O$ for a given input set I, and an output element $o \in O$, *provenance* should identify the input subset $I^* \subseteq I$ containing those elements that contributed to o's derivation. First we define provenance for each transformation type, then we show how this "one-level" provenance is used to derive workflow provenance.

Provenance for single transformations is straightforward and intuitive:

- **Map Provenance.** Given a map function M, the provenance of an output element $o \in M(I)$ is the input element i that produced o, i.e., $o \in M(\{i\})$.
- **Reduce Provenance.** Given a reduce function R, the provenance of an output element $o \in R(I)$ is the group $G_j \subseteq I$ that produced o, i.e., $o \in R(G_j)$.
- **Union Provenance.** Given a union transformation U, the provenance of an output element $o \in U(I_1, \ldots, I_m) = I_1 \cup \ldots \cup I_m$ is the corresponding input element i in some I_k, where $i = o$. (Recall from Section 2.2 that $\cup$ is guaranteed to be a disjoint union.)
- **Split Provenance.** Given a split transformation S where $S(I) = (O_1, \ldots, O_r)$ and $I = O_1 \cup \ldots \cup O_r$, the provenance of an output element $o \in O_k$ is the corresponding element $i \in I$, where $i = o$.

The provenance of an output subset $O^* \subseteq O$ is simply the union of the provenance for all elements $o \in O^*$.

Now suppose we have a GMRW, and we would like the provenance of an output element in terms of the initial inputs to the workflow. For our recursive definition, we more generally define the provenance of any data element involved in the workflow—input, intermediate, or output.

DEFINITION 3.1 (GMRW PROVENANCE). Consider a GMRW W with initial inputs $I_1, \ldots, I_m$ and any data element e. The provenance of e in W, denoted $P_W(e)$, is an m-tuple $(I_1^*, \ldots, I_m^*)$, where $I_1^* \subseteq I_1, \ldots, I_m^* \subseteq I_m$. If e is an initial input element, i.e., $e \in I_k$, then $P_W(e) = \{e\}$. Otherwise, let T be the transformation that output e. Let $P_T(e)$ be the one-level provenance of e with respect to T as defined above. Then $P_W(e) = \bigcup_{e' \in P_T(e)} P_W(e')$. □

Having defined GMRW provenance in the intuitive way, we would like to make sure it gives us something meaningful. Specifically, we desire the following "replay" property.

PROPERTY 3.1 (REPLAY PROPERTY). Consider an output element o, and let $P_W(o) = (I_1^*, \ldots, I_m^*)$ be the provenance of o in workflow W. If we run $I_1^*, \ldots, I_m^*$ through W, denoted $W(I_1^*, \ldots, I_m^*)$, then o is part of the result: $o \in W(I_1^*, \ldots, I_m^*)$. □

The replay property holds for our running example and for a very large class of GMRWs, but unfortunately it does not hold all the time. Suppose our running example is changed in the following two ways (shown in Figure 3):

- **TweetScan** may output more than one element when a tweet discusses multiple movies, i.e., **TweetScan** is now one-many.
- Output GoodMovies (GM) is input to an additional reduce function **CountByRating**, which emits the number of movies for each good median rating 6–10.

Using the modified workflow, here is a scenario where the replay property does not hold. Suppose Tweets consists of three tweets: tweet t_1 produces ratings (*Inception*,8) and (*Twilight*,8); tweet t_2 produces rating (*Twilight*,2); tweet t_3 produces rating (*Twilight*,5). Let Diggs be empty. Dropping the 1000 ratings requirement, for these input data sets, output RatingCount contains (rating:8,count:1) based on *Inception* with median rating 8, while output BadMovies contains (*Twilight*) with median rating 5.

Based on Definition 3.1, for the output element o = (rating:8,count:1) in RatingCount, we get $P_W(o) = \{t_1\}$, which agrees with our intuition since t_1 contains all of the elements in Tweets related to those movies with a median rating of 8 (just *Inception*). However, suppose we reran the workflow on o's provenance, i.e., using tweet t_1 only. The result in output RatingCount is the "incorrect" value (rating:8,count:2). Only one of the three ratings for *Twilight* is used, therefore its median is also computed as 8. In terms of our formalism, $o \notin W(P_W(o))$.

Let us try to understand what characteristics of the example workflow caused the replay property to be violated. When **TweetScan**

is rerun on $P_W(o) = \{t_1\}$, it produces an element e = (*Twilight*,8) that is irrelevant to the provenance of the output element we're interested in. Such extraneous elements can be produced only by one-many map or many-many reduce functions. When reduce function **Aggregate** is run on the two elements produced by tweet t_1, the correct median (*Inception*,8) is produced, but so is incorrect median (*Twilight*,8), since not all data for *Twilight* is being processed by the workflow. The incorrect median wouldn't be harmful on its own, but when it is combined with the correct median in the **CountByRating** transformation, an incorrect output is produced. Note that if either reduce function **Aggregate** or **CountByRating** were monotonic, the problem would not have occurred: If **Aggregate** were monotonic, it could not produce an incorrect output value, since it is operating on a subset of the correct input. If **CountByRating** were monotonic, then extra input could only create additional output, not eliminate the correct output.

It turns out that the specific pattern of three (or more) transformations with certain properties, as exhibited by the above example, is the *only* case in which rerunning a workflow on the provenance of an output element o is not guaranteed to produce o. We prove the following theorem in Appendix A.

THEOREM 3.1. Consider a GMRW W composed of transformations $T_1, \ldots, T_n$, with initial inputs $I_1, \ldots, I_m$. Let o be any output element, and consider $P_W(o) = (I_1^*, \ldots, I_m^*)$.

1. If all map and reduce functions in W are one-one or many-one, respectively, then $o = W(I_1^*, \ldots, I_m^*)$. (Note this result is stronger than the general $o \in W(I_1^*, \ldots, I_m^*)$.)
2. If there is at most one nonmonotonic reduce function in W, then $o \in W(I_1^*, \ldots, I_m^*)$. □

In fact, for the replay property $o \in W(I_1^*, \ldots, I_m^*)$ to be violated, the one-many map or many-many reduce function must precede the two nonmonotonic reduce functions in the workflow.

For workflows not satisfying Theorem 3.1, the recursive definition of provenance still yields an intuitive result. However, we believe it is important to be able to rerun a workflow on an output element's provenance—and get the output element in the result—as part of the use of provenance for debugging purposes. (Provenance-based *selective refresh* [18], comprised of backward tracing followed by forward propagation, requires a similar property. Our approach to selective refresh for arbitrary workflows in [18] would deem this example workflow "unsafe" and disallow it.) In the GMRW context, we can automatically augment any ill-behaved workflow W with extra filters that ensure $o \in W(P_W(o))$ for any output element o. This result is formalized in the following Corollary, proved in Appendix B.

COROLLARY 3.1. Consider a GMRW W composed of transformations $T_1, \ldots, T_n$, with initial inputs $I_1, \ldots, I_m$. Let o be any output element, and consider $P_W(o) = (I_1^*, \ldots, I_m^*)$. Let W^* be constructed from W by replacing all nonmonotonic reduce functions T_j with $T_j \circ \sigma_j$, where σ_j is a filter that removes all elements from the output of T_j that were not in the output of T_j when $W(I_1, \ldots, I_m)$ was run originally.[1] Then $o \in W^*(P_W(o))$. □

4. PROVENANCE CAPTURE & TRACING

In this section we describe, at an abstract level, how provenance according to our definitions can be captured and stored during workflow execution. We also give algorithms for forward and backward tracing. The next section will give details of our actual implementation as an extension to Hadoop. For now let us assume that all input, intermediate, and output data sets are persistent, although we will see in Section 5 that our Hadoop implementation discards certain intermediate data sets.

Capturing and storing provenance according to our definitions is straightforward: For map functions, we extract the unique identifier (ID) of each input element that produces one or more output elements, and we add that ID to each of the output elements. For reduce functions, we keep track of the grouping key for each input group, and we add that key to each output element produced by the group. In Section 5, we describe in more detail how our Hadoop implementation wraps map and reduce functions automatically to emit these extra fields during execution. Recall that we introduced union and split operations in Section 2.2 for the purposes of analysis. In reality, these operations are incorporated into map and reduce functions and do not affect our capture and storage scheme.

Now consider backward tracing. The following algorithm implements the recursive definition of provenance given in Section 3.

ALGORITHM 4.1 (BACKWARD TRACING). Consider a GMRW W with initial inputs $I_1, \ldots, I_m$. Recursive function *backward_trace* returns the provenance of a set E of data elements from a single input, intermediate, or output data set:

backward_trace$(E, W, \{I_1, \ldots, I_m\})$:
if $E \subseteq I_k$ for $1 \leq k \leq m$ **then return** E;
else { $T \leftarrow$ transformation that output the set containing E;
  if T is a map function
    then $E' \leftarrow$ input elements to T with ID that annotates an element in E;
  if T is a reduce function
    then $E' \leftarrow$ input elements to T with grouping key that annotates an element in E;
  $E'_1, \ldots E'_n \leftarrow E'$ partitioned by input sets;
  $I^* \leftarrow \emptyset$;
  for $i = 1..n$ **do**
    $I^* \leftarrow I^* \cup$ *backward_trace*$(E'_i, W, \{I_1, \ldots, I_m\})$;
  return I^*; }}

For forward tracing, the overall algorithm is simply the converse of backward tracing:

ALGORITHM 4.2 (FORWARD TRACING). Consider a GMRW W with final outputs $O_1, \ldots, O_r$, and any set E of data elements from a single input, intermediate, or output data set. Algorithm *forward_trace* returns the output elements derived from any element in E:

forward_trace$(E, W, \{O_1, \ldots, O_r\})$:
  if $E \subseteq O_k$ for $1 \leq k \leq r$ **then return** E;
  else { $T \leftarrow$ transformation that processes E;
    if T is a map function
      then $E' \leftarrow$ output elements from T with ID corresponding to an element in E;
    if T is a reduce function
      then $E' \leftarrow$ output elements from T with grouping key corresponding to an element in E;
    $E'_1, \ldots E'_n \leftarrow E'$ partitioned by output sets;
    $O^* \leftarrow \emptyset$;
    for $i = 1..n$ **do**
      $O^* \leftarrow O^* \cup$ *forward_trace*$(E'_i, W, \{O_1, \ldots, O_r\})$;
    return O^*;}

[1] We assume all intermediate/output data sets have been stored for provenance-tracing purposes; see Section 4.

5. RAMP SYSTEM

We have built a system called *RAMP* (*Reduce And Map Provenance*) implementing provenance for GMRWs as described in this paper. RAMP is built as an extension to Hadoop [3]. Because the basic execution unit in Hadoop is a *MapReduce job* consisting of one map function followed by one reduce function, in the workflows supported by RAMP, each transformation is a MapReduce job. Specifically, a GMRW is implemented as a *MapReduce workflow* in RAMP:

1. If the GMRW contains a map function followed by a reduce function, the two transformations are treated as a single MapReduce job in RAMP. In particular, no intermediate data is stored between the map and reduce functions.
2. Map functions in the GMRW that are not followed by a reduce function are treated as a MapReduce job without a reduce function.
3. Reduce functions in the GMRW that are not preceded by a map function are treated as a MapReduce job with the identity map function.

In other respects, RAMP captures and traces provenance as discussed in Section 4.

RAMP's approach to provenance capture is wrapper-based and transparent to Hadoop, retaining Hadoop's parallel execution and fault tolerance. Furthermore, users need not be aware of provenance capture while writing MapReduce jobs—wrapping is automatic, and RAMP stores provenance separately from the input and output data.

When input and output data sets are stored in files, RAMP provides efficient default schemes for assigning element IDs and storing provenance; these schemes are described in Section 5.3. For other settings, RAMP allows users to define custom ID and storage schemes.

We discuss how RAMP performs provenance capture and tracing in Sections 5.1 and 5.2, respectively. These sections do not assume RAMP's default implementation for file input and output, which is discussed in Section 5.3.

5.1 Provenance Capture

Recall from Section 4 that all intermediate data between transformations is stored, and provenance is captured one transformation at a time. As described above, in the MapReduce workflows supported by RAMP, each transformation is a MapReduce job. Thus, this section describes how provenance is captured for a single MapReduce job. Section 5.2 explains how RAMP uses the captured provenance to support provenance tracing through arbitrary workflows comprised of multiple MapReduce jobs.

In Hadoop, all data elements are assumed to be key/value pairs. When running a MapReduce job consisting of a map function and a reduce function, the map output elements are grouped by their key before being processed by the reduce function. Otherwise, keys are simply part of the data.

Hadoop users supply the following five components to define a MapReduce job [5]:

- *Record-reader:* Reads the input data and parses it into input key/value pairs for the mapper.
- *Mapper:* Defines the map function.
- *Combiner:* Defines partial aggregation by key (optional).
- *Reducer:* Defines the reduce function.
- *Record-writer:* Writes output key/value pairs from the reducer in a specified output format.

RAMP implements the provenance capture scheme from Section 4 by wrapping all of these components. For presentation purposes, we consider MapReduce jobs without a combiner; the extension for combiners is straightforward. We also assume all MapReduce jobs do have a reducer; RAMP's extension for map-only jobs is similarly straightforward.

5.1.1 Map functions

For map functions, RAMP adds to each map output element a unique ID for the input element that generated the output element. Specifically, RAMP annotates the *value* part of the map output element, allowing Hadoop to correctly group the map output elements by key for the reduce function.

The following procedure specifies how RAMP wraps the record-reader and mapper to perform ID annotation (Figure 4).

PROCEDURE 5.1 (WRAPPING A MAP FUNCTION).

1. The record-reader assigns a unique ID p to the input element (k^i, v^i) that it emits.
2. The record-reader's wrapper emits $(k^i, \langle v^i, p\rangle)$.
3. The mapper's wrapper takes $(k^i, \langle v^i, p\rangle)$ as input and feeds (k^i, v^i) to the mapper.
4. For each mapper output (k^m, v^m), the mapper's wrapper emits $(k^m, \langle v^m, p\rangle)$. □

No provenance is actually stored at this point; provenance storage is performed by the wrapped reducer and record-writer, as explained next.

5.1.2 Reduce functions

For reduce functions, RAMP stores the reduce provenance as a mapping from a unique ID for each output element (k^o, v^o) to the grouping key k^m that produced (k^o, v^o). It simultaneously stores the map provenance as a mapping from the grouping key k^m to the input element ID p_j's. By storing map provenance after the map output elements have been grouped, RAMP allows all input element IDs corresponding to the same grouping key to be stored together. Since the grouping key k^m merely joins the map and reduce provenance, k^m is replaced with an integer ID k^m_{ID}.

The following procedure describes how RAMP wraps the reducer and record-writer (Figure 5).

PROCEDURE 5.2 (WRAPPING A REDUCE FUNCTION).

1. The reducer's wrapper iterates over all annotated map output elements $(k^m, \langle v^m_j, p_j\rangle)$'s with the same key k^m and feeds each (k^m, v^m_j) to the reducer.
2. While feeding the reducer, the reducer's wrapper stores the map provenance (k^m_{ID}, p_j)'s.
3. For each reducer output (k^o, v^o), the reducer's wrapper emits $(k^o, \langle v^o, k^m_{ID}\rangle)$ to the record-writer's wrapper.
4. The record-writer's wrapper takes $(k^o, \langle v^o, k^m_{ID}\rangle)$ as input and feeds (k^o, v^o) to the record-writer.
5. The record-writer assigns a unique ID q to the output element (k^o, v^o) that it writes.
6. The record-writer's wrapper stores the reduce provenance (q, k^m_{ID}). □

An alternative scheme for provenance would be to store the input element ID p_j's directly for each output element ID. However, this scheme wastes space when the reduce function is many-many. Moreover, we cannot implement this scheme efficiently in Hadoop: we would need to collect all ID p_j's in advance by iterating over

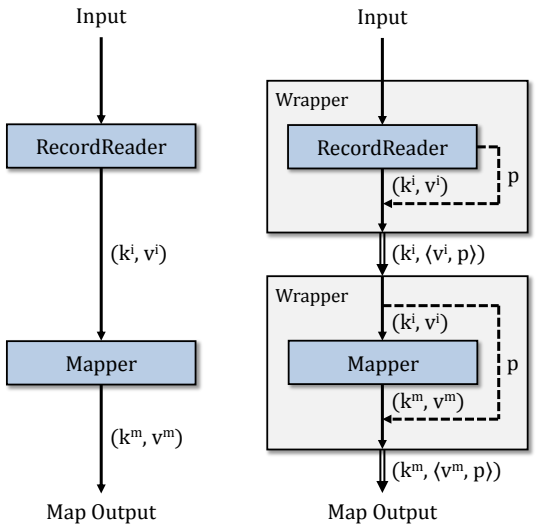


Figure 4: Wrapping a map function with RAMP.

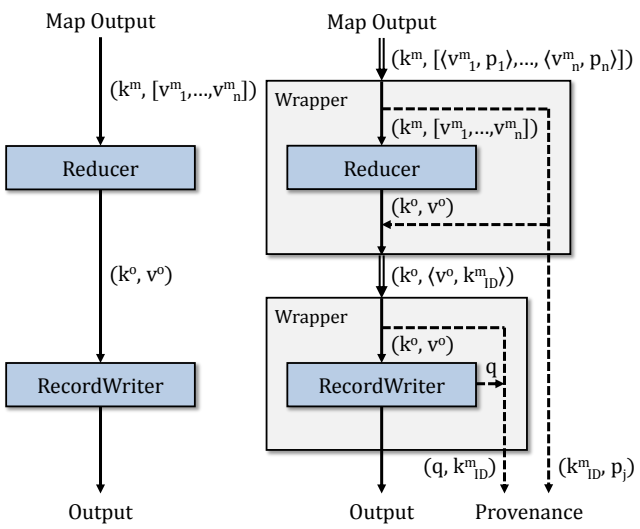


Figure 5: Wrapping a reduce function with RAMP.

all map output elements $(k^m, \langle v_j^m, p_j \rangle)$'s because the reducer can produce an output element (k^o, v^o) before all ID p_j's are seen by our wrapper. In contrast, RAMP stores the map and reduce provenance independently, joining them later during provenance tracing. One disadvantage of RAMP's scheme is that extraneous provenance data may be written if the reduce function does not produce any output elements for a particular grouping key.

5.2 Provenance Tracing

Implementing the *backward_trace* function of Algorithm 4.1 in the RAMP system is fairly straightforward, although our tracing scheme has not yet been made as efficient as possible.

Since each RAMP transformation is a MapReduce job as described above, a single backward tracing step for one output element proceeds as follows:

1. Given an output element ID q, RAMP accesses the reduce provenance as specified in the previous section to determine the corresponding grouping key ID k_{ID}^m.
2. Using k_{ID}^m, RAMP accesses the map provenance as specified in the previous section to retrieve all relevant input element ID p_j's.

The IDs returned by step 2 can either be used to fetch actual data elements, or they can be fed to recursive invocations of backward tracing until the initial input data sets are reached. Our current implementation traces recursively one element at a time, however we would expect efficiency to improve in some cases by tracing multiple elements together.

Notice that RAMP's provenance capture scheme is biased towards backward tracing, which we assume is a more frequent operation than forward tracing. In the forward-tracing setting, we are given a set of input element IDs, and we need to find all output elements that corresponds to these input element IDs. Without auxiliary structures, each forward-tracing step would require a complete scan of the map provenance, which is not sorted on input element IDs. Thus, as a first step to facilitate forward tracing, we certainly need to build indexes on the input element ID field. As will be seen in Section 6, our performance experiments have also focused on backward tracing. Enabling efficient forward tracing and measuring its performance is a next step in the RAMP system.

5.3 Data Sets in Files

Consider the specific workflow setting where all input, intermediate, and output data sets are stored in files, as is typical using Hadoop. In this setting, RAMP uses (*filename*, *offset*) as a default unique ID for each data element. For input element IDs, RAMP maps each filename to an integer ID, maintaining a dictionary with the actual filenames. For output element IDs, RAMP is able to omit the filename and use the offset alone: each provenance data file is associated with a particular output data file. Variable-length encoding for integers allows RAMP to implement this element ID scheme efficiently in terms of space overhead.

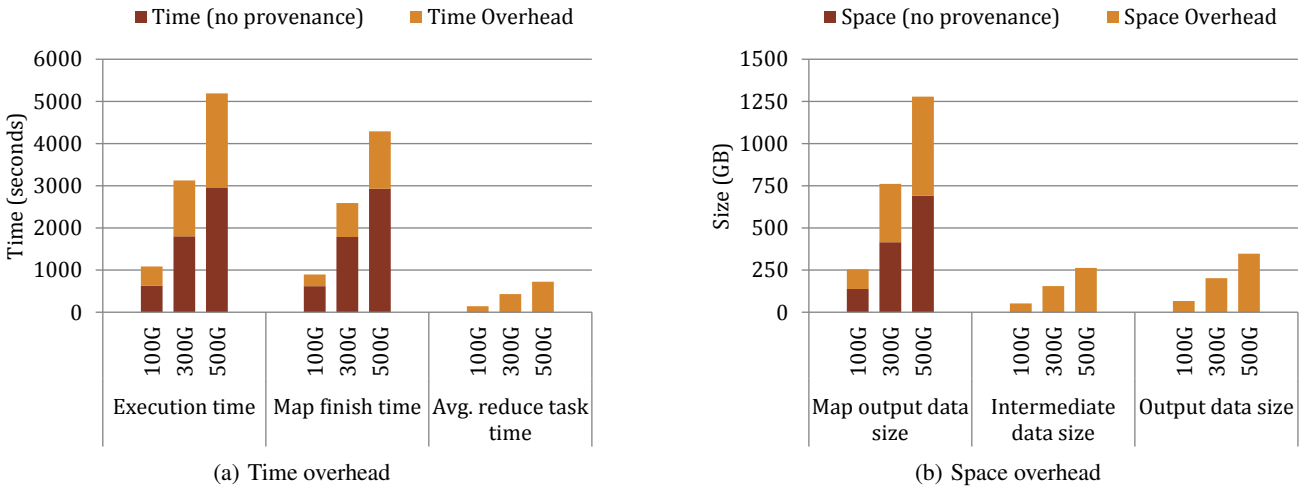
Notice that our scheme enables efficient backward tracing without special indexes: Output element IDs, which are the element's offset in the output data file, increase as each output element is appended. Thus, reduce provenance is automatically stored in ascending key order. Exploiting this order, RAMP performs binary search on the provenance data during backward tracing.

6. EXPERIMENTAL EVALUATION

We present performance experiments conducted using RAMP on two MapReduce workflows, each consisting of a single MapReduce job. (Since our experiments are focused more on capture than on tracing, a single MapReduce job is sufficient.) The two MapReduce jobs in our experiments were *Wordcount* and *Terasort* [21], included in the Hadoop distribution. Wordcount counts the number of occurrences of each word in a set of input text files; we used 100, 300, and 500 GB of input text generated randomly from 8000 distinct English words. Terasort sorts 100-byte records stored in files; we used 10^9, 3×10^9, and 5×10^9 random records as input (93, 279, and 466 GB respectively).

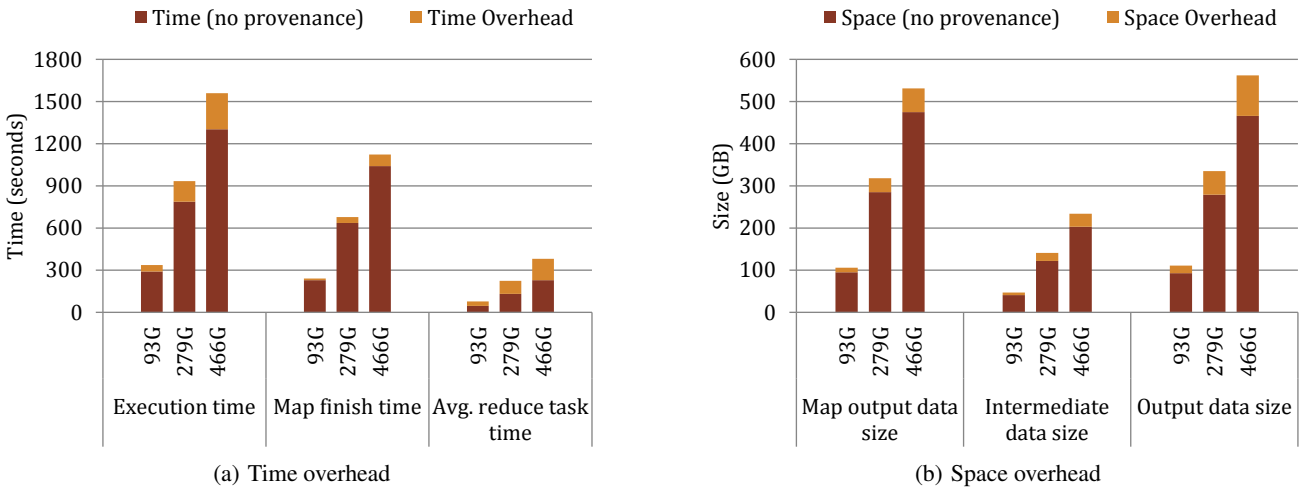
Note that Wordcount and Terasort are very different in terms of multiplicity. Wordcount is a many-one transformation, with a huge fan-in for large input data sizes. On the other hand, Terasort is a one-one transformation. We discuss how the multiplicity affects the time and space overhead of provenance capture in Section 6.1.

The cluster we used for our experiments consisted of 51 large Amazon Elastic Compute Cloud (EC2) instances, each with 7.5 GB memory, two virtual cores with 2 EC2 Compute Units each, and 850 GB instance storage. We launched all instances with 64-bit



(a) Time overhead (b) Space overhead

Figure 6: Overhead of provenance capture in Wordcount.



(a) Time overhead (b) Space overhead

Figure 7: Overhead of provenance capture in Terasort.

Amazon Linux AMI and installed Java 1.6.0_22 and the modified version of Hadoop 0.21.0. One instance served as the master node and acted as both *name node* and *job tracker*; the other 50 instances served as slave nodes. Each slave node was allowed to run two map tasks and two reduce tasks concurrently, and the number of reduce tasks was set to 100. We configured Hadoop following the guidelines for real-world cluster configurations [4]. The changes from the default configuration included increasing the heap size for task JVMs to 1 GB, compressing map output using LZO [25], and allowing task JVMs to be reused. We also increased the sort buffer size so that the entire output from each map task fit in the buffer without spilling to disk. Finally, the replication factor for output files was set to 1.

Our performance results are summarized as follows:

- We first measured time and space overhead of provenance capture. For our two experiments, provenance capture incurred 20-76% time overhead. For Terasort, the space overhead incurred by provenance capture was 21%; for Wordcount, the space overhead can be made arbitrarily large. Details are reported in Section 6.1.
- We then measured the time to backward-trace output elements after provenance has been captured. Backward-tracing one element took as little as 1.5 seconds in Terasort, but again can be made arbitrarily large in Wordcount. Details are reported in Section 6.2.

6.1 Performance: Capture

We report the time and space overhead associated with capturing provenance in our experiments. For each input data size, we ran Wordcount five times and Terasort three times, with and without capturing provenance; we report the average of the trials, which had little variance.

Figures 6(a) and 7(a) report the time overhead by showing the time taken without provenance capture (dark bar) and the additional time with capture (light bar).

- *Execution time:* Time for the entire MapReduce job to complete.
- *Map finish time:* Time until all map tasks are complete.
- *Average reduce task time:* Average time for an individual reduce task.

Figures 6(b) and 7(b) similarly report the space overhead of provenance.

- *Map output data size:* The size of the map output data (before applying the combiner).
- *Intermediate data size:* The size of the intermediate data after applying the combiner and LZO compression.
- *Output data size:* The size of the final output data.

Observe in Figures 6 and 7 that time and space overhead are closely related. A larger output data size increases the average reduce task time, because the output data is written by the reduce tasks. Similarly, a larger intermediate data size increases the map finish time as well as the average reduce task time, because map tasks store intermediate data temporarily in local disk, and reduce tasks sort the intermediate data set. In our experiments, the map output data size had little impact on the execution time, because the entire map output data set fit in the sort buffer.

For Wordcount, provenance capture incurred 72-76% time overhead (Figure 6(a)). Because Wordcount is a many-one transformation, each intermediate and output data element is annotated with many input element IDs. Moreover, the number of annotations per data element increases with input size, because we have the same fixed number of words across all data sizes. As a result, the space overhead percentage grows linearly with input size, and can be made arbitrarily large. The significant increase in both intermediate and output data sizes (shown in Figure 6(b), where the dark bars are not even visible) correlates with the large time overhead of provenance capture.

For Terasort, provenance capture incurred 16-20% time overhead and 19-21% space overhead (Figures 7(a) and 7(b), respectively). Because Terasort is a one-one transformation, each intermediate and output data element is annotated with exactly one input element ID. The moderate increase in both intermediate and output data sizes (shown in Figure 7(b)) is consistent with the small time overhead.

6.2 Performance: Tracing

For both experiments, we measured the time to backward-trace output elements after provenance has been captured, for varying input data sizes. In our experiments we did not fetch the actual input data elements as part of tracing; we just identified their element IDs. Overall, we have not yet focused our work on making backward tracing as efficient as possible; we report the tracing performance based on our current implementation.

For Wordcount, tracing one element took approximately 1, 3, and 5 minutes, for 100, 300, and 500 GB input data sizes respectively. Note that even when we trace a single output element, the data sizes processed become quite large: For 100 GB input data, the average number of occurrences for each word is about 1,289,000. As discussed in the previous section, because we have a fixed number of words across all input data sizes, the provenance of individual output elements grows linearly with input size. This behavior explains the linear growth of tracing time.

For Terasort, tracing one element took approximately 1.5 seconds for all input data sizes. Since the data sizes processed are very small—each element traced produces one element as a result—the binary search (Section 5.3) tends to dominate tracing time. We suspect that deploying an appropriate index could improve backward tracing time by at least factor of two. The use of indexes in general is an immediate area of future work.

7. CONCLUSIONS AND FUTURE WORK

This paper defines provenance for map and reduce functions, and it identifies properties that hold when one-level provenance is applied recursively in arbitrary GMRWs. We have built a prototype system as an extension to Hadoop that supports provenance capture and tracing; performance numbers are reported.

As described in Section 5, implementing efficient backward and forward tracing is an important next step in the RAMP system. For both types of tracing, building appropriate indexes will be a key component of our approach. In addition, we plan to measure the performance of provenance capture and tracing for MapReduce workflows consisting of multiple jobs. We have thus far successfully captured provenance for MapReduce workflows compiled by Pig [20] using RAMP. For future experiments, the PigMix [6] benchmarks seem like a good starting point.

One obvious general avenue for future work is to incorporate SQL processing into our workflows. SQL nodes interspersed with map and reduce functions can form a rich and interesting environment. Some SQL queries are map or reduce functions already, allowing them to slot right into our framework. Other SQL queries may not fit the map or reduce paradigm precisely, but do have known, well-understood provenance [11, 13] that can be incorporated via extensions to our framework. Finally, several recent systems (e.g., Hive [24]) compile SQL queries into MapReduce jobs. We intend to compare provenance captured and traced using previous methods for SQL against using our system on the compiled GMRWs.

8. REFERENCES

[1] The Open Provenance Model — Core Specification (v1.1). Dec. 2009. http://eprints.ecs.soton.ac.uk/18332/.

[2] M. K. Anand, S. Bowers, and B. Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, 2010.

[3] Apache. Hadoop. *http://hadoop.apache.org/*.

[4] Apache. Hadoop cluster setup. *http://hadoop.apache.org/common/docs/r0.21.0/cluster_setup.html*.

[5] Apache. Mapreduce tutorial. *http://hadoop.apache.org/mapreduce/docs/r0.21.0/mapred_tutorial.html*.

[6] Apache. Pigmix benchmarks. *http://wiki.apache.org/pig/PigMix*.

[7] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, 2008.

[8] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1), 2005.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[10] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[11] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[12] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1), 2003.

[13] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2), 2000.

[14] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[16] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[17] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD*, 2008.

[18] R. Ikeda, S. Salihoglu, and J. Widom. Provenance-based refresh in data-oriented workflows. Technical report, Stanford University InfoLab, March 2010.

[19] P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. Goble. Data lineage model for Taverna workflows with lightweight annotation requirements. In *IPAW*, 2008.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[21] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. *http://l.yimg.com/a/i/ydn/blogs/hadoop/yahoo2009.pdf*, 2009.

[22] S. Sarawagi. Information extraction. *Found. Trends databases*, 1:261–377, March 2008.

[23] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3), 2005.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[25] K. Weil. hadoop-lzo. *http://www.github.com/kevinweil/hadoop-lzo/*.

[26] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.

[27] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *SIGMOD*, 2007.

APPENDIX

A. PROOF OF THEOREM 3.1

Theorem: Consider a GMRW W composed of transformations $T_1, \ldots, T_n$, with initial inputs $I_1, \ldots, I_m$. Let o be any output element, and consider $P_W(o) = (I_1^*, \ldots, I_m^*)$.

1. If all map and reduce functions in W are one-one or many-one, respectively, then $o = W(I_1^*, \ldots, I_m^*)$. (Note this result is stronger than the general $o \in W(I_1^*, \ldots, I_m^*)$.)
2. If there is at most one nonmonotonic reduce function in W, then $o \in W(I_1^*, \ldots, I_m^*)$. □

A.1 Proof of Part 1

We prove a stronger property: Let O be the output of W and let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$. The proof is by induction on the structure of W.

Base case $W = M$ where M is a map function. By definition, $P_M(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_M(o_j))$. For $j = 1..n$, $P_M(o_j) = \{i_j\}$ such that $M(\{i_j\}) = \{o_j\}$. By the definition of map functions, and since M is one-one, $M(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(M(i_j)) = \{o_1, ..., o_n\}$.

Base case $W = R$ where R is a reduce function. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $R(G_j) = \{o_j\}$. By the definition of reduce functions, and since R is many-one, $R(G_1 \cup \ldots \cup G_n) = \bigcup_{j=1}^{n}(R(G_j)) = \{o_1, ..., o_n\}$.

Base case $W = U$ where U is a union transformation. U has input data sets $I_1, \ldots, I_m$. By definition, $P_U(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_U(o_j))$. For $j = 1..n$, $P_U(o_j) = \{i_j\}$ where i_j is the element in some I_k that corresponds to o_j. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1, \ldots, I_m$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_m)$, where each $I'_k = \mathbb{I} \cap I_k$. Then $U(\{i_j\}) = \{o_j\}$. By the definition of union transformations, $U(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(U(\{i_j\})) = \{o_1, ..., o_n\}$.

Base case $W = S$ where S is a split transformation. S has output sets $O_1, \ldots, O_r$. By definition, $P_S(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_S(o_j))$. For $j = 1..n$, o_j is in some O_k, and $P_S(o_j) = \{i_j\}$ such that $S(\{i_j\}) = O'_1, \ldots, O'_r$, where $O'_k = \{o_j\}$ and $O'_h = \emptyset$ for $h \neq k$. Since split transformations are context-independent on each element (Section 2.2), $S(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(S(\{i_j\})) = \{o_1, ..., o_n\}$.

Now suppose workflows $W'_1, \ldots, W'_p$ satisfy the inductive hypothesis: $W'(P_{W'}(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$ for any $o_1, ..., o_n$ in the output of W'. Consider an additional transformation T and the workflow W that is constructed by making the outputs of $W'_1, \ldots, W'_p$ the inputs of T. (For all T other than union transformations, $p = 1$.) We use $\circ$ for workflow composition.

Map: Suppose $W = W' \circ M$. Let $P_M(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. Since M is one-one, by the definitions of map provenance and map functions, $M(\{o'_1, ..., o'_n\}) = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) = \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$.

Reduce: Suppose $W = W' \circ R$. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. Since R is many-one, by the definitions of reduce provenance and reduce functions, $R(G_1 \cup ... \cup G_n) = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(G_1 \cup \ldots \cup G_n)) = G_1 \cup \ldots \cup G_n$. Thus $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$.

Union: Suppose W is composed of $W'_1, \ldots, W'_p$ followed by U. U has input data sets $I_1^U, \ldots, I_p^U$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1^U, \ldots, I_p^U$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I_k^U$. By the definitions of union provenance and union transformations, $U(\{o'_1, ..., o'_n\}) = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) = \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$.

Split: Suppose $W = W' \circ S$. S has output sets $O_1, \ldots, O_r$. Let $P_S(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definitions of split provenance and split transformations, $S(\{o'_1, ..., o'_n\}) = O'_1, \ldots, O'_r$, where $O'_1 \cup \ldots \cup O'_r = \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) = \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) = \{o_1, ..., o_n\}$. □

A.2 Proof of Part 2

Lemma 1: Consider a GMRW W with output O. Suppose there are no nonmonotonic reduce functions in W. Let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Proof: By induction on the structure of W.

Base case $W = M$ where M is a map function. By definition, $P_M(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_M(o_j))$. For $j = 1..n$, $P_M(o_j) = \{i_j\}$ such that $o_j \in M(\{i_j\})$. By the definition of map functions, $M(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(M(i_j)) \supseteq \{o_1, ..., o_n\}$.

Base case $W = R$ where R is a reduce function. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $o_j \in R(G_j)$. By the definition of reduce functions, $R(G_1 \cup \ldots \cup G_n) = \bigcup_{j=1}^{n}(R(G_j)) \supseteq \{o_1, ..., o_n\}$.

Base case $W = U$ where U is a union transformation. U

has input data sets $I_1, \ldots, I_m$. By definition, $P_U(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_U(o_j))$. For $j = 1..n$, $P_U(o_j) = \{i_j\}$ where i_j is the element in some I_k that corresponds to o_j. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1, \ldots, I_m$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_m)$, where each $I'_k = \mathbb{I} \cap I_k$. Then $U(\{i_j\}) = \{o_j\}$. By the definition of union transformations, $U(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(U(\{i_j\})) = \{o_1, ..., o_n\}$.

Base case $W = S$ where S is a split transformation. S has output sets $O_1, \ldots, O_r$. By definition, $P_S(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_S(o_j))$. For $j = 1..n$, o_j is in some O_k. $P_S(o_j) = \{i_j\}$ such that $S(\{i_j\}) = O'_1, \ldots, O'_r$, where $O'_k = \{o_j\}$ and $O'_h = \emptyset$ for $h \neq k$. Since split transformations are context-independent on each element, $S(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(S(\{i_j\})) = \{o_1, ..., o_n\}$.

Now suppose workflows $W'_1, \ldots, W'_p$ satisfy the inductive hypothesis: $W'(P_{W'}(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$ for any $o_1, ..., o_n$ in the output of W'. Consider an additional transformation T and the workflow W that is constructed by making the outputs of $W'_1, \ldots, W'_p$ the inputs of T.

Map: Suppose $W = W' \circ M$. Let $P_M(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definitions of map provenance and map functions, if $I' \supseteq \{o'_1, ..., o'_n\}$, then $M(I') \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) \supseteq \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Reduce: Suppose $W = W' \circ R$. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ...\cup G_n)$, where each group G_j produces o_j. Since R is monotonic, if $I' \supseteq G_1 \cup ... \cup G_n$, then $R(I') \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(G_1 \cup \ldots \cup G_n)) \supseteq G_1 \cup \ldots \cup G_n$. Thus $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Union: Suppose W is composed of $W'_1, \ldots, W'_p$ followed by U. U has input data sets $I^U_1, \ldots, I^U_p$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I^U_1, \ldots, I^U_p$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I^U_k$. By the definitions of union provenance and union transformations, if $I' \supseteq \{o'_1, ..., o'_n\}$, then $U(I') \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) \supseteq \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$.

Split: Suppose $W = W' \circ S$. S has output sets $O_1, \ldots, O_r$. Let $P_S(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definitions of split provenance and split transformations, if $I' \supseteq \{o'_1, ..., o'_n\}$, then $S(I') = O'_1, \ldots, O'_r$ where $O'_1 \cup \ldots \cup O'_r \supseteq \{o_1, ..., o_n\}$. By the inductive hypothesis, $W'(P_{W'}(\{o'_1, ..., o'_n\})) \supseteq \{o'_1, ..., o'_n\}$. Thus, $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. □

Lemma 2: Consider a GMRW W with output O. Suppose there are no nonmonotonic reduce functions in W. Let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Proof: By induction on the structure of W.

Base case $W = M$ where M is a map function. Let M have input set I and output set O. Let $P_M(\{o_1, ..., o_n\}) = \{i_1, ..., i_n\}$. By the definition of map functions, $\{i_1, ..., i_n\} \subseteq I$, and $M(\{i_1, ..., i_n\}) \subseteq M(I) = O$.

Base case $W = R$ where R is a reduce function. Let R have input set I and output set O. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $G_j \subseteq I$. Since R is monotonic and $G_1 \cup \ldots \cup G_n \subseteq I$, $R(G_1 \cup \ldots \cup G_n) \subseteq R(I) = O$.

Base case $W = U$ where U is a union transformation. U has input data sets $I_1, \ldots, I_m$. By definition, $P_U(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_U(o_j))$. For $j = 1..n$, $P_U(o_j) = \{i_j\}$ where i_j is the element in some I_k that corresponds to o_j. For any set $\mathbb{I}$ that combines subsets of U's input sets $I_1, \ldots, I_m$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_m)$, where each $I'_k = \mathbb{I} \cap I_k$. Then $U(\{i_j\}) = \{o_j\}$. By the definition of union transformations, $U(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(U(\{i_j\})) = \{o_1, ..., o_n\} \subseteq O$.

Base case $W = S$ where S is a split transformation. S has output sets $O_1, \ldots, O_r$. By definition, $P_S(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_S(o_j))$. For $j = 1..n$, o_j is in some O_k. $P_S(o_j) = \{i_j\}$ such that $S(\{i_j\}) = O'_1, \ldots, O'_r$, where $O'_k = \{o_j\}$ and $O'_h = \emptyset$ for $h \neq k$. Since split transformations are context-independent on each element, $S(\{i_1, ..., i_n\}) = \bigcup_{j=1}^{n}(S(\{i_j\})) = \{o_1, ..., o_n\} \subseteq O$.

Now suppose workflows $W'_1, \ldots, W'_p$ satisfy the inductive hypothesis: $W'(P_{W'}(\{o_1, ..., o_n\})) \subseteq O'$ for any $o_1, ..., o_n$ in O', where O' is the output of W'. Consider an additional transformation T and the workflow W that is constructed by making the outputs of $W'_1, \ldots, W'_p$ the inputs of T.

Map: Suppose $W = W' \circ M$. Let $P_M(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definition of map provenance, $\{o'_1, ..., o'_n\} \subseteq O'$. Let O^* denote $W'(P_{W'}(\{o'_1, ..., o'_n\}))$. By the inductive hypothesis, $O^* \subseteq O'$. By the definition of map functions, since $O^* \subseteq O'$, $M(O^*) \subseteq M(O') = O$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Reduce: Suppose $W = W' \circ R$. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. By the definition of reduce provenance, $G_1 \cup \ldots \cup G_n \subseteq O'$. Let O^* denote $W'(P_{W'}(G_1 \cup \ldots \cup G_n))$. By the inductive hypothesis, $O^* \subseteq O'$. Since R is monotonic and $O^* \subseteq O'$, $R(O^*) \subseteq R(O') = O$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Union: Suppose W is composed of $W'_1, \ldots, W'_p$ followed by U. U has input data sets $I^U_1, \ldots, I^U_p$. Let $P_U(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definition of union provenance, $\{o'_1, ..., o'_n\} \subseteq O'$. For any set $\mathbb{I}$ that combines subsets of U's input sets $I^U_1, \ldots, I^U_p$, let $U(\mathbb{I})$ denote $U(I'_1, \ldots, I'_p)$, where each $I'_k = \mathbb{I} \cap I^U_k$. Let O^* denote $W'(P_{W'}(\{o'_1, ..., o'_n\}))$. By the inductive hypothesis, $O^* \subseteq O'$. By the definition of union transformations, since $O^* \subseteq O'$, $U(O^*) \subseteq U(O') = O$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$.

Split: Suppose $W = W' \circ S$. S has output sets $O_1, \ldots, O_r$. Let $P_S(\{o_1, ..., o_n\}) = \{o'_1, ..., o'_n\}$. By the definition of split provenance, $\{o'_1, ..., o'_n\} \subseteq O'$. Let O^* denote $W'(P_{W'}(\{o'_1, ..., o'_n\}))$. By the inductive hypothesis, $O^* \subseteq O'$. By the definition of O', $S(O') = O_1, \ldots, O_r$. Let $S(O^*) = O^*_1, \ldots, O^*_r$. By the definition of split transformations, since $O^* \subseteq O'$, each $O^*_j \subseteq O_j$. Thus, $W(P_W(\{o_1, ..., o_n\})) \subseteq O$. □

Theorem: We prove a stronger property: Let $o_1, \ldots, o_n$ be elements of O. Then $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. The proof is by induction on the structure of W. The base case and inductive step proofs are exactly the same as those for Lemma 1, with the exception of the Reduce inductive step.

Inductive step for Reduce: Suppose $W = W' \circ R$. If R is monotonic, then the Reduce inductive step proof from Lemma 1 suffices. Suppose R is nonmonotonic. Then W' has no nonmonotonic reduce functions. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. Let O^* denote $W'(P_{W'}(G_1 \cup \ldots \cup G_n)$. Lemma 1 on W' tells us that $O^* \supseteq G_1 \cup \ldots \cup G_n$. Let O' denote the output of W'. Lemma 2 on W' tells us that $O^* \subseteq O'$. By the definition of reduce provenance, each group G_j is equal to the set of all elements in O' with G_j's key. Since $G_j \subseteq O^* \subseteq O'$, there cannot be any element in $O^* - G_j$ that has G_j's key. Thus, each group G_j is equal to the set of all elements in O^* with G_j's key. $R(O^*) \supseteq \bigcup_{j=1}^{n} R(G_j)$, which implies $W(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. □

B. PROOF OF COROLLARY 3.1

Corollary: Consider a GMRW W composed of transformations $T_1, \ldots, T_n$, with initial inputs $I_1, \ldots, I_m$. Let o be any output element, and consider $P_W(o) = (I_1^*, \ldots, I_m^*)$. Let W^* be constructed from W by replacing all nonmonotonic reduce functions T_j with $T_j \circ \sigma_j$, where σ_j is a filter that removes all elements from the output of T_j that were not in the output of T_j when $W(I_1, \ldots, I_m)$ was run originally. Then $o \in W^*(P_W(o))$.

Proof: We prove a stronger property: Let O be the output of W and let $o_1, ..., o_n$ be elements of O. Then $O \supseteq W^*(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. This property clearly implies the corollary. The proof is by induction on the structure of W. The base case and inductive step proofs follow directly from the analogous cases in Lemmas 1 and 2 of Appendix A, with the exceptions of nonmonotonic reduce functions.

Base case $W = R$ where R is a nonmonotonic reduce function. By definition, $P_R(\{o_1, ..., o_n\}) = \bigcup_{j=1}^{n}(P_R(o_j))$. For $j = 1..n$, $P_R(o_j) = G_j$ such that $o_j \in R(G_j)$. By the definition of reduce functions, $R(G_1 \cup \ldots \cup G_n) = \bigcup_{j=1}^{n}(R(G_j)) \supseteq \{o_1, ..., o_n\}$. Let σ_R be the filter associated with R. Since $\{o_1, \ldots, o_n\} \subseteq O$, no element in $\{o_1, \ldots, o_n\}$ is removed by σ_R. Thus, $\sigma_R(R(G_1 \cup \ldots \cup G_n)) \supseteq \{o_1, ..., o_n\}$. Since σ_R filters only elements not in O, $O \supseteq (R \circ \sigma_R)(G_1 \cup \ldots \cup G_n)$.

Inductive step for Reduce: Suppose $W = W' \circ R$ where R is nonmonotonic. Let $P_R(\{o_1, ..., o_n\}) = (G_1 \cup ... \cup G_n)$, where each group G_j produces o_j. Let O^* denote $W'^*(P_{W'}(G_1 \cup \ldots \cup G_n)$. Let O' be the output of W'. By the inductive hypothesis, $O' \supseteq O^* \supseteq (G_1 \cup \ldots \cup G_n)$.

By the definition of reduce provenance, each group G_j is equal to the set of all elements in O' with G_j's key. Since $G_j \subseteq O^* \subseteq O'$, there cannot be any element in $O^* - G_j$ that has G_j's key. Thus, each group G_j is equal to the set of all elements in O^* with G_j's key. $R(O^*) \supseteq \bigcup_{j=1}^{n} R(G_j) \supseteq \{o_1, \ldots, o_n\}$. Let σ_R be the filter associated with R. Since $\{o_1, \ldots, o_n\} \subseteq O$, no element in $\{o_1, \ldots, o_n\}$ is removed by σ_R. Thus, $W^*(P_W(\{o_1, ..., o_n\})) \supseteq \{o_1, ..., o_n\}$. Since σ_R is the final step of W^*, and σ_R filters only elements not in O, $O \supseteq W^*(P_W(\{o_1, ..., o_n\}))$. □

DBToaster: Agile Views in a Dynamic Data Management System

Oliver Kennedy*
EPFL
oliver.kennedy@epfl.ch

Yanif Ahmad
Johns Hopkins University
yanif@jhu.edu

Christoph Koch
EPFL
christoph.koch@epfl.ch

ABSTRACT

This paper calls for a new breed of lightweight systems – *dynamic data management systems (DDMS)*. In a nutshell, a DDMS manages large *dynamic data structures* with *agile, frequently fresh views*, and provides a facility for monitoring these views and triggering application-level events. We motivate DDMS with applications in large-scale data analytics, database monitoring, and high-frequency algorithmic trading. We compare DDMS to more traditional data management systems architectures. We present the DBToaster project, which is an ongoing effort to develop a prototype DDMS system. We describe its architecture design, techniques for high-frequency incremental view maintenance, storage, scaling up by parallelization, and the various key challenges to overcome to make DDMS a reality.

1. INTRODUCTION

Dynamic, continuously evolving sets of records are a staple of a wide variety of today's data management applications. Such applications range from large, social, content-driven Internet applications, to highly focused data processing verticals like data intensive science, telecommunications and intelligence applications. There is no one brush with which we can paint a picture of all dynamic data applications – they face a broad spectrum of update volumes, of update impact on the body of data present, and data freshness requirements. However, modern data management systems either treat updates and their impact on datasets and queries as an afterthought, by extending DBMS with triggers and heavyweight views [17, 33, 47, 48], or only handle small, recent sets of records in data stream processing [12, 25, 34, 37].

We propose dynamic data management systems (DDMS) that are capable of handling arbitrary, high-frequency or high-impact updates on a general dataset, and any standing queries (views) posed on the dataset by long-running applications and services. We base the design of DDMS on four criteria:

1. The stored dataset is large and changes frequently.
2. The maintenance of materialized views dominates ad-hoc querying.
3. Access to the data is primarily by monitoring the views and performing simple computations on top of them. Some updates cause events, observable in the views, that trigger subsequent computations, but it is rare that the data store is accessed asynchronously by humans or applications.
4. Updates happen primarily through an *update stream*. Computations triggered by view events typically do not cause updates: there is usually no feedback loop.

A DDMS is a lightweight system that provides large dynamic data structures to support declarative views of data. A DDMS is *agile*, keeping internally maintained views fresh in the face of dynamic data. Client applications primarily interact with a DDMS by registering callbacks for view changes, rather than by accessing views directly. A DDMS does not necessarily provide additional DBMS functionality such as persistency, transactions, or recoverability.

Compared to a classical DBMS, a DDMS differs in its reaction to updates. To minimize response times, updates must be performed immediately upon arrival, precluding bulk processing. This determines the programming model: compared to a DBMS, control flow is reversed, and the DDMS invokes application code, not vice versa.

An active DBMS [36] could simulate a DDMS through triggers, but is not optimized for such workloads, and even if support for state-of-the-art incremental view maintenance is present, performs very poorly. Thus, DDMS differ from active databases in their being optimized for different workloads. DDMS are optimized for event processing and monitoring tasks, while active database systems are optimized to support traditional DBMS functionality such as transactions, which are not necessarily present in DDMS.

Compared to a data stream processing system and particularly an event processing system (such as Cayuga [25], SASE+ [1]), DDMS have much larger states, which will usually have to be maintained in secondary storage, and require drastically different query processing techniques. In a stream processor, the queries reside in the system while the data streams by. In a DDMS on the other hand, the data state is maintained in the system while a stream of updates passes through (much more like an OLTP system).

Moreover, event and stream processors [12, 34, 37] support drastically different query languages which are designed to ensure that only very small state has to be maintained, using windows or constructs from formal language theory [44].

*Also Dept. of Computer Science, Cornell University.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA. .

DDMS views are often rather complex and expensive, including large non-windowed joins and aggregation. In general, we expect DDMS to support standard SQL. The query processing techniques most suitable for such workloads come from DBMS research – incremental view maintenance in particular – and update stream research [16] but do not scale to high-frequency view maintenance.

We present two classes of applications motivating the desiderata and design choices of a DDMS, and then discuss one application scenario in detail.

Large-scale data analytics, but not as a batch job. Large-scale data analytics in the cloud are mostly performed on massively parallel processing engines such as map/reduce. These systems are not databases, as some strata of the systems, scientific computing, and large-scale Web applications communities find important to emphasize. Nevertheless, our research community can play an important role in making such systems more useful and effective. Clearly, the last word on posing *queries* in such systems has not been said.

Map/reduce-like systems achieve scalability at the cost of response time and interactivity. However, there is an increasing number of important applications of large-scale analytics that call for more interactivity or better response times that allow for online use. Among large Web applications, examples include (social or other) network monitoring and statistics [31], search with interactive feedback [7], interactive recommendations, keeping personalized Web pages at social networking sites up to date [46], and so forth. Many of these applications are not yet mission-critical to Web applications companies, but are becoming increasingly necessary for establishing and maintaining a competitive advantage.

Large-scale data analytics is equally present in more classical business applications such as data warehousing and scientific applications. Take the case of data warehousing with real-time updates: as data warehouses become increasingly mission-critical to commercial and scientific enterprises, the importance of up-to-date analyses increases. Traditionally, OLAP systems are not optimized for frequent updating, and may be considerably out-of-date. A DDMS could dramatically improve the freshness of warehoused data.

A DDMS is well-suited for use in large-scale data analytics through its provision of large dynamic data structures as views, instead of forcing programmers to re-implement view computations manually on top of key-value stores, and its emphasis on simple lightweight systems as opposed to the use of monolithic DBMS. Continually fresh DDMS views at first seem at odds with the bulk update processing dogma of large scale analytics systems, but enable important applications that require interactivity or event processing.

Database monitoring. There is an ever-increasing set of use cases in which aggregate views over large databases need to be continuously maintained and monitored as the database evolves. These queries can be thought of as continuous queries on the stream of updates to a database. However, it is only moderately helpful to view this as a stream processing scenario since the queries depend on very large state (the database) rather than a small window of the update stream, and cannot be handled by data stream processing systems.

Examples include policy monitoring (e.g., to comply with regulatory requirements to monitor databases of financial institutions, say to detect money laundering schemes) [6], network security monitoring, aiming to detect sophisticated attacks that span extended time periods, and data-driven simulations such as Markov-Chain Monte Carlo (MCMC). MCMC is an extremely powerful paradigm for probabilistic databases [15]. Query processing with MCMC involves walking between database states, iteratively making local alterations to the database. Each database state encountered while doing this is considered a sample; a view is re-evaluated and statistics on its results are collected. The key technical database problem is thus to compute the view for as many samples as possible, as quickly as possible [45]. This is precisely the kind of workload that DDMS are designed for. Further examples of database monitoring can be found in certain forms of automated trading. In the following, we discuss one form of data-driven automated trading which may well prove to be a killer application for DDMS.

Algorithmic trading with order books. In recent years, algorithmic trading systems have come to account for a majority of volume traded at the major US and European financial markets (for instance, for 73% of all US equity trading volume in the first quarter of 2009 [20]). The success of automated trading systems depends critically on strategy processing speeds: trading systems that react faster to market events tend to make money at the cost of slower systems. Unsurprisingly, algorithmic trading has become a substantial source of business for the IT industry; for instance, it is the leading vertical among the customer bases for high-speed switch manufacturers (e.g., Arista [43]) and data stream processing.

A typical algorithmic trading system is run by mathematicians who develop trading strategies and by programmers and systems experts who implement these strategies to perform fast enough, using mainly low-level programming languages such as C. Developing trading strategies requires a feedback loop of simulation, back-testing with historical data, and strategy refinement based on the insights gained. This loop, and the considerable amount of low-level programming that it causes, is the root of a very costly *productivity bottleneck*: in fact, the number of programmers often exceeds the number of strategy designers by an order of magnitude.

Trading algorithms often perform a considerable amount of data crunching that could in principle be implemented as SQL views, but cannot be achieved by DBMS or data stream processing systems today: DBMS are not able to (1) *update their views at the required rates* (for popular stocks, hundreds of orders per second may be executed, even outside burst times) and stream engines are not able to (2) *maintain large enough data state* and support suitable query languages (non-windowed SQL aggregates) on this state. A data management system fulfilling these two requirements would yield a very substantial productivity increase that can be directly monetized – the holy grail of algorithmic trading.

To understand the need to maintain and query a large data state, note that many stock exchanges provide a detailed view of the market microstructure through complete bid and ask *limit order books*. The bid order book is a table of purchase offers with their prices and volumes, and correspondingly the ask order book indicates investors' selling orders. Exchanges execute trades by matching bids and asks by price and favoring earlier timestamps. Investors continually add, modify or withdraw limit orders, thus one may view order books as relational tables subject to high update

volumes. The availability of order book data has provided substantial opportunities for automatic algorithmic trading.

EXAMPLE 1.1. To illustrate this, we describe the Static Order Book Imbalance (SOBI) trading strategy. SOBI computes a volume-weighted average price (VWAP) over those orders whose volume makes up a fixed upper k-fraction of the total stock volume in both bid and ask order books. SOBI then compares the two VWAPs and, based on this, predicts a future price drift (for example a bid VWAP larger than an ask VWAP indicates demand exceeds supply, and prices may rise). For simplicity, we present the VWAP for the bids only:

```
select avg(b2.price * b2.volume) as bid_vwap
from   bids b2
where  k * (select sum(volume) from bids)
         > (select sum(volume) from bids b1
            where b1.price > b2.price);
```

Focusing on the k-fraction of the order book closest to the current price makes the SOBI strategy less prone to attacks known as *axes* (large tactical orders far from the current price that will thus not be executed but may confuse competing algorithms).

Given continuously maintained views for VWAP queries on bid and ask order books, an implementation of the SOBI strategy only takes a few lines of code that trigger a buy or sell order whenever the ratio between the two VWAPs exceeds a certain threshold. □

We return to the two desiderata for query engines for algorithmic trading pointed out above. For trading algorithms to be successful, (1) views such as VWAP need to be maintained and monitored by the algorithms at or close to the trading rate. However, (2) the views cannot be expressed through time-, row- or punctuation-based window semantics. This lends weight to the need for DDMS that support agile views on large, long-lived state.

Structure of the paper. There are many technical challenges in making a DDMS a reality. This paper initiates a study of DDMS and presents DBToaster, a prototype developed by the authors. The contributions of this paper are as follows: In Section 2, we further characterize the notion of a DDMS and discuss a state machine abstraction of DDMS that further clarifies the programming model and abstractions relevant to users of such systems. Section 3 presents the DBToaster view maintenance technique. It demonstrates how the state machine abstraction, which calls for the compilation and aggressive pre-computation of the state transition function (viewed differently, this is the set of update triggers that cause views to be refreshed), leads to new incremental query evaluation techniques. Section 4 discusses storage management in DBToaster. Section 5 discusses scaling up through parallelization. We conclude in Section 6.

2. DDMS ARCHITECTURE

We now examine the architecture of a DDMS, as illustrated in Figure 1. The core component of a DDMS is its runtime engine. Unlike a traditional database system where the same engine manages all database instances, each individual DDMS execution runtime is constructed around a

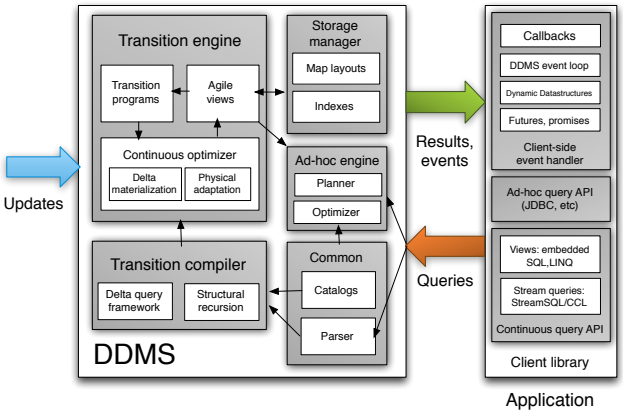


Figure 1: Dynamic Data Management System (DDMS) and Application Interface Architecture

specific set of queries provided by the client program (e.g., via SQL code embedded inline in the program), each defining an *agile view*.

2.1 Application Interfaces

The data that is processed by a DDMS arrives at the system in the form of an update stream of tuple insertions, deletions and modifications. The stream need not be ordered in any shape or form, and deletions are assumed to apply to tuples that have already been seen at some arbitrary prior point on the stream. Updates are fully processed on-the-fly, and their effects on agile views are realised in atomic fashion, prior to working on any subsequent update. Depending on the type of results requested by queries, any results arising from updates will be directly forwarded to application code as agile views are maintained.

DBToaster provides a wide variety of client interfaces to issue queries and obtain results from the DDMS, to reflect the diverse needs of applications built on top of our tool. Today's stream processors tend to be black-box systems that run completely decoupled from the application. Client libraries interact with stream processors through remote procedure call abstractions, issuing queries and new data through function calls, and either polling or being notified whenever results appear on a queue that is associated with a TCP socket connected to the stream processor.

In DBToaster, the set of agile views requested by clients, the *visible schema*, forms the primary read interface between client programs and the DDMS runtime. Clients can submit queries for which the DDMS materializes an agile view through three methods: (1) an embedded language, whose syntax and data model are natural fits to the host language in which the client application is written. Examples include embedded SQL, and collection comprehension oriented approaches such as LINQ, Links, and Ferry [11, 18, 29]. One interesting challenge with the embedded language approach is that of enabling asynchronous event-driven programming. Whereas language embeddings are natural for ad-hoc querying, we have yet to see these approaches for stream processing. This is the main mode of specifying queries that we focus on in this paper. (2) a continuous query client API, as done with existing stream client libraries, which sends a query string to the DDMS server for parsing, compilation, and agile view construction. The query string may be specified in a standard streaming language such as StreamSQL

or CCL [21]. The client may specify several ways to receive results, as seen below. (3) an ad-hoc query client API, which issues a one-time query to the DDMS, and returns the agile view as a datastructure to be used by the remainder of the client program. This API may be used in both synchronous and asynchronous modes, as indicated by the type of result requested. The query is specified in standard SQL.

Given these modes of issuing queries to the DDMS, our client interface supports four methods of receiving results: (1) callbacks, which can be specified as handlers as part of the continuous query API. Callbacks receive a stream of query results, and are the simplest form of result handlers that run to completion on query result events. (2) a DDMS event loop, which multiplexes result streams for multiple queries. Applications may register callbacks to be executed on any result observed on the event loop, allowing complex application behavior through dynamic registration, observation and processing of results on the event loop. (3) dynamic datastructures, which are read-only from the application perspective. The datastructure appears as a native collection type in the host language, facilitating natural access for the remainder of the program. Ad-hoc queries use this method for results by default. Continuous queries may also use this method in which case the datastructure acts as a proxy with accessors that pull in any updates from the DDMS when invoked. (4) promises and futures [26], which provide a push-based proxy datastructure for the result. A future is an object whose value is not initially known and is provided at a later time. A program using a query returning a future can use the future as a native datatype, in essence constructing a client-side dataflow to be executed whenever the future's value is bound. In our case, this occurs whenever query results arrive from the DDMS. Language embedded stream processing can be supported by futures, or program transformations to construct client side dataflow, such as continuation passing style as found in the programming languages literature [40].

2.2 DDMS Internals

The internals of the runtime engine itself are best viewed through the lens of a state machine. Compared to similar abstractions for complex event processors [1, 25], the state is substantially larger. Conceptually, the state represents an entire relational database and transitions represent changes in the base relations: events in the update stream.

Compiling transitions. Each transition causes maintenance work for our agile views, and just as with incremental view maintenance, this work can be expressed as queries. Maintenance can be aided by dynamic data structures, that is, additional agile views making up the *auxiliary schema*. A DDMS is a long-running system, operating on a finite number of update streams. This combination of characteristics naturally suggests *compiling* and specializing the runtime for each transition and associated maintenance performed by a DDMS. The transition compiler generates lightweight transition programs that can be invoked by the runtime engine with minimal overhead on the arrival of events. We describe the compiler in further detail in Section 3.

Storage management and ad-hoc query processing. Given the instantiation of an auxiliary schema and agile views, a DDMS must intelligently manage memory utilization, and the memory-disk boundary as needed. The storage manager of a DDMS is responsible for the efficient representation of both the agile views and any index structures required on these views. Section 4 discusses the issue of indexing, as well as how views are laid out onto disk. Supporting ad-hoc query processing turns out to be relatively straightforward given that the core of a DDMS continuously maintains agile views. Ad-hoc queries can be rewritten to use agile views in a similar fashion to the materialized view usage problem in standard query optimization. A key challenge here is how to ensure consistency, such that ad- hoc queries do not use inconsistent agile views as updates stream in and the DDMS performs maintenance. On the other hand, we do not want ad-hoc queries to block the DDMS' maintenance process and incur result delivery latency for continuous queries. One option here is to maintain a list of undo actions for each ad-hoc query with respect to agile view maintenance. This design is motivated by the fact that continuous queries are the dominant mode of usage, and ad-hoc queries are expected to occur infrequently, thus we bias the concurrency control burden towards ad-hoc queries.

Runtime adaptivity. Significant improvements in just-in-time (JIT) compilation techniques means that transition programs need not be rigid throughout the system's lifetime. A DDMS includes a compiler and optimizer working in harmony, leveraging update stream statistics to guide the decisions to be made across the database schema, state and storage. For example, the compiler may choose to compute one or more views on the fly, rather than maintaining it in order to keep expected space usage within predefined bounds. The optimizer's decisions are made in terms of the space being used, the cost of applying transitions on updates, as well as information from a storage manager that aids in physical aspects of handling large states, including implementing a variety of layouts and indexes to facilitate processing.

3. REALIZING AGILE VIEWS

Agile views are database views that are maintained as incrementally as possible. Despite more than three decades of research into incremental view maintenance (IVM) techniques [17, 35, 47, 48], agile views have not been realised, and one of our key contributions in handling large dynamic datasets is to exploit further opportunities for incremental computation during maintenance. Conceptually, current IVM techniques use delta queries for maintenance. Our observation is that the delta query is itself a relational query that is amenable to incremental computation. We can materialize delta queries as auxiliary views, and recursively determine deltas of delta queries to maintain these auxiliary views. Furthermore repeated delta transformations successively simplify queries.

3.1 View Maintenance in DBToaster

We present our observation in more detail and with an example. Given a query q defining a view, IVM yields a pair $\langle m, Q' \rangle$, where m is the materialization of q, and Q' is a set of delta queries responsible for maintaining m (one for each relation used in q that may be updated). DBToaster makes the following insight regarding IVM: current IVM algorithms evaluate a delta query entirely from scratch on every update to any relation in q, using standard query processing techniques. DBToaster exploits that a delta query q' from set Q' can be incrementally computed using the same principles as for the view query q, rather than evaluated in full.

To convey the essence of the concept, IVM takes q, pro-

duces $\langle m, Q' \rangle$ and performs m `+=` $q'(u)$ at runtime, where u is an update to a relation R and q' is the delta query for updates to R in Q'. We call this one step of *delta (query) compilation*. This is the extent of query transformations applied by IVM for incremental processing of updates. DBToaster applies this concept *recursively*, transforming queries to *higher-level deltas*. DBToaster starts with q, produces $\langle m, Q' \rangle$ and then recurs, taking each q' to produce $\langle m', Q'' \rangle$ and repeating. Here, each m' is maintained as m' `+=` $q''(v)$, where v is also an update, (possibly) different from u above, and q'' is the delta query from Q'' for the relation being updated. We refer to q' and q'' as first- and second-level delta queries respectively. We again recur for each q'', materialize it as m'', maintain it using third-level queries Q''', and so forth.

While delta queries are relational queries, they have certain characteristics that facilitate recursive delta compilation. First, DBToaster delta queries are parameterized SQL queries (with the same notion of parameter as in, say, Embedded SQL), with parameter values taken from updates. Thus, in particular, higher-level deltas are just (parameterized) SQL queries, but are not *higher-order* in the sense of functional programming as some queries in complex-value query languages are [8].

To illustrate parameters, we apply one step of delta compilation on the following query q over a schema `R(a int, b int), S(b int, c int)`:

$q =$ `select sum(a*c) from R natural join S`

For an update u that is an insertion of tuple $\langle$`@a`, `@b`$\rangle$ into relation R, the delta for q is:

$$q_R = \Delta_u(q) = \texttt{select sum(@a*c) from values(@a,@b), S where S.b = @b}$$
$$= \texttt{@a*(select sum(c) from S where S.b=@b)}$$

The `values (...)` clause is PostgreSQL syntax for a singleton relation defined in the query. Transforming a query into its delta form for an update u on R introduces parameters in place of R's attributes. We also apply a rewrite exploiting distributivity of addition and multiplication to factor out parameter `@a` from the query.

The second property that is key to making recursive delta processing feasible is that, for a large class of queries, delta queries are structurally strictly simpler than the queries that the delta queries are taken off. This can be made precise as follows. Consider SQL queries that are sum-aggregates over positive relational algebra. Consider positive relational algebra queries as unions of select-project-join (SPJ) queries. The *degree* of an SPJ query is the number of relations joined together in it. The degree of a positive relational algebra query is the maximum of the degrees of its member SPJ queries and the degree of an aggregate query is the degree of its positive relational algebra component. The rationale for such a formalization – based on viewing queries as polynomials over relation variables – is discussed in detail in [23]. It is proven in that paper that the delta query of a query of degree k is of degree $\max(k-1, 0)$. A delta query of degree 0 only depends on the update but not on the database relations. So DBToaster guarantees that a k-th level delta query $q^{(k)}$ has lower degree than a *(k-1)*-th level query $q^{(k-1)}$. Recursive compilation terminates when all conjuncts have degree zero.

Consider the delta query q_R above, which is of degree 1 while q is of degree 2. Query q_R is simpler than q since it does not contain the relation R. We can further illustrate the point by looking at a recursive compilation step on q_R. The second compilation step materializes q_R as:

$m_R =$ `select sum(c) from S where S.b=@b`

omitting the parameter `@a` since it is independent of the above view definition query. DBToaster can incrementally maintain m_R with the following delta query on an update v that is an insertion of tuple $\langle$`@c`, `@d`$\rangle$ into relation S:

$q_{RS} = \Delta_v(q_R) =$ `select @c from values(@c,@d)`

The delta query q_{RS} above has degree zero since its conjuncts contain no relations, indeed the query only consists of parameters. Thus recursive delta compilation terminates after two rounds on query q. In this compilation overview, we have not discussed the maintenance code for views m, m_R, and m_{RS} to allow the reader to focus on the core recursive compilation and termination concepts. We now discuss the data structures used to represent auxiliary materialized views, and then provide an in-depth example of delta query compilation including all auxiliary views created and the code needed to maintain these views.

Agile Views. DBToaster materializes higher-level deltas as agile views for high-frequency update applications with continuous group-by aggregate query workloads. Agile views are represented as main memory (associative) map data structures with two sets of keys (that is a doubly-indexed map $m[\vec{x}][\vec{y}]$), where the keys can be explained in terms of the delta query defining the map.

As we have mentioned, delta queries are parameterized SQL queries. The first set of keys (the *input* keys) correspond to the parameters, and the second set (the *output* keys) to the select-list of the defining query. In the event that a parameter appears in an equality predicate with a regular attribute, we omit it from the input keys because we can unify the parameter. We briefly describe other interesting manipulations of parameterized queries in our framework in the following section, however a formal description of our framework is beyond the scope of this paper.

Example. Figure 2 shows the compilation of a query q:

```
select l.ordkey, o.sprior, sum(l.extprice)
from   Customer c, Orders o, Lineitem l
where  c.custkey = o.custkey and l.ordkey = o.ordkey
group by l.ordkey, o.sprior
```

inspired by TPC-H Query 3, with a simplified schema:

```
Customer(custkey,name,nationkey,acctbal)
Lineitem(ordkey,extprice)
Order(custkey,ordkey,sprior)
```

The first step of delta compilation on q produces a map m. The aggregate for each group $\langle ordkey, sprior \rangle$ can be accessed as $m[][ordkey, sprior]$. We can answer query q by iterating over all entries (groups) in map m, and yielding the associated aggregate value. The first step also computes a delta query q_c by applying standard delta transformations as defined in existing IVM literature [17, 35, 47, 48]. In summary, these approaches substitute a base relation in a query with the contents of an update, and rewrite the query. For example, on an insertion to the `Customer` relation, we can substitute this relation with an update tuple $\langle$`@ck,@nm,@nk,@bal`$\rangle$:

Input (parent query)		Update	Output: auxiliary map, delta query		
$q =$	`select l.ordkey, o.sprior, sum(l.extprice) from Customer c, Orders o, Lineitem l where c.custkey = o.custkey and l.ordkey = o.ordkey group by l.ordkey, o.sprior;`	`+Customer (ck,nm,nk,bal)`	$m[][ordkey, sprior]$	$q_c =$	`select l.ordkey, o.sprior, sum(l.extprice) from Orders o, Lineitem l where @ck = o.custkey and l.ordkey = o.ordkey group by l.ordkey, o.sprior;`
q_c:	Recursive call, see previous output	`+Lineitem (ok,ep)`	$m_c[][custkey, ordkey, sprior]$	$q_{cl} =$	`select @ok, o.sprior,@ep*sum(1) from Orders o where @ck = o.custkey and @ok = o.ordkey`
q_{cl}:	Recursive call, see previous output	`+Order (ck2,ok2,sp)`	$m_{cl}[][custkey, ordkey, sprior]$	$q_{clo} =$	`select @sp, count() where @ck = @ck2 and @ok = @ok2;`

Figure 2: Recursive query compilation in DBToaster. For query q, we produce a sequence of materializations and delta queries for maintenance: $\langle m, q' \rangle, \langle m', q'' \rangle, \langle m'', q''' \rangle$. This is a partial compilation trace, our algorithm considers all permutations of updates.

```
select l.ordkey, o.sprior, sum(l.extprice)
from   values (@ck,@nm,@nk,@bal)
         as c(custkey,name,nationkey,acctbal),
       Orders o, Lineitem l
where  c.custkey = o.custkey and l.ordkey = o.ordkey
group by l.ordkey, o.sprior
```

Above the substitution replaces the `Customer` relation with a singleton set consisting of an update tuple with its fields as parameters. We can simplify q_c as:

```
q_c =  select   l.ordkey, o.sprior,sum(l.extprice)
       from    Orders o, Lineitem l
       where    @ck = o.custkey
       and      l.ordkey = o.ordkey
       group by l.ordkey, o.sprior;
```

The query rewrite replaces instances of attributes with parameters through variable substitution, as well as more generally (albeit not seen in this example for simpler exposition of the core concept of recursive delta compilation), exploiting unification, and distributivity properties of joins and sum aggregates to factorize queries [23].

This completes one step of delta compilation. Our compilation algorithm also computes deltas to q for insertions to `Order` or `Lineitem` (i.e. q_o and q_l). We list the full transition program for all insertions at the end of the example (deletions are symmetric, and ommitted due to space limitations). IVM techniques evaluate q_c on every insertion to `Customer`. To illustrate the recursive nature of our technique, we walk through the recursive compilation of q_c to m_c, q_{cl} on an insertion to `Lineitem` (see the second row of Figure 2). At this second step, DBToaster materializes q_c with its parameter `@ck` and group-by fields as $m_c[][custkey, ordkey, sprior]$, and uses this map m_c to maintain the query view m:

```
on_insert_customer(ck,nm,nk,bal):
  m[][ordkey,sprior] += m_c[][ck,ordkey,sprior];
```

As it turns out, all maps instantiated from simple equijoin aggregate queries such as TPCH Query 3 have no input keys. Maps with input keys only occur as a result of inequality predicates and correlated subqueries, for example the VWAP query from Section 1.

Above, we have a trigger statement in a C-style language firing on insertions to the `Customer` relation, describing the maintainence of m by reading the entry $m_c[\texttt{ck}, ordkey, sprior]$ instead of evaluating $q_c(\texttt{ck}, Orders, Lineitem)$. Notice that the trigger arguments do not contain *ordkey* or *sprior*, so where are these variables defined? In DBToaster, this statement implicitly performs an iteration over the domain of the map being updated. That is, map m is updated by looping over all $\langle ordkey, sprior \rangle$ entries in its domain, invoking lookups on m_c for each entry and the trigger argument `ck`. Map read and write locations are often (and for a large class of queries, always) in one-to-one correspondence, allowing for an embarrassingly parallel implementation (see Section 5). For clarity, the verbose form of the statement is:

```
on_insert_customer(ck,nm,nk,bal):
  for each ordkey,sprior in m:
    m[][ordkey,sprior] += m_c[][ck,ordkey,sprior];
```

Throughout this document we use the implicit loop form. Furthermore, this statement is never implemented as a loop, but relies on a map data structure supporting partial key access, or *slicing*. This is trivially implemented with secondary indexes for each partial access present in any maintenance statement, in this case a secondary index yielding all $\langle ordkey, sprior \rangle$ pairs for a given `ck`. This form of maintenance statement is similar in structure to the concept of *marginalization* in probability distributions, essentially the map m is a marginalization of map m_c over the attribute `ck`, for each `ck` seen on the update stream.

Returning to the delta q_{cl} produced by the second step of compilation, we show its derivation and simplification below:

```
select l.ordkey, o.sprior,
       sum(l.extprice)          select @ok, o.sprior,
from   Orders o, values                 @ep*sum(1)
       (@ok,@ep) as         =>  from   Orders o
       l(ordkey,extprice)       where  @ck = o.custkey
where  @ck = o.custkey          and    @ok = o.ordkey
and    l.ordkey = o.ordkey
```

Notice that q_{cl} has a parameter `@ck` in addition to the substituted relation `Lineitem`. This parameter originates from the attribute `c.custkey` in q, highlighting that map parameters can be passed through multiple levels of compilation. The delta q_{cl} is used to maintain the map m_c on insertions to `Lineitem`, and is materialized in the third step of compilation as $m_{cl}[][custkey, ordkey, sprior]$. The resulting maintenance code for m_c is (corresponding to Line 8 of the full listing):

```
on_insert_lineitem(ok,ep) :
  m_c[][custkey, ok, sprior] +=
    ep * m_cl[][custkey, ok, sprior];
```

Above, we iterate over each $\langle custkey, sprior \rangle$ pair in the map m_c, for the given value of the trigger argument `ok`. Note we have another slice access of m_{cl}, of $\langle custkey, sprior \rangle$ pairs for a given `ok`. The third step of recursion on insertion to

`Order` is the terminal step, as can be seen on inspection of the delta query q_{clo}:

```
select o.sprior, count()
from values (@ck2,@ok2,@sp)
as o(custkey,ordkey,sprior) =>
where @ck = o.custkey
and   @ok = o.ordkey
```

```
select @sp, count()
where @ck = @ck2
and   @ok = @ok2
```

In the result of the simplification, the delta q_{clo} does not depend on the database since it contains no relations, only parameters. Thus the map m_{cl} can be maintained entirely in terms of trigger arguments and map keys alone. Note this delta contains parameter equalities. These predicates constrain iterations over map domains, for example the maintenance code for q_{clo} would be rewritten as:

```
on_insert_order(ck2,ok2,sp) :
  m_cl[][ck, ok, sp] +=
    if ck==ck2 && ok==ok2
    then 1 else 0;
```

```
=> m_cl[][ck2, ok2, sp]
     += 1;
```

where, rather than looping over map m_{cl}'s domain and testing the predicates, we only update the map entry corresponding to `ck2`, `ok2` from the trigger arguments.

We show the trigger functions generated by DBToaster below, for all possible insertion orderings, including for example deltas of q on insertions to `Order` and then `Lineitem`. We express the path taken as part of the map name as seen for m_c and m_{cl} in our walkthrough. Some paths produce maps based on equivalent queries; DBToaster detects these and reuses the same map. Due to limited space, we omit the case for deletions, noting that these are symmetric to insertions except that they decrement counts.

```
1.  on_insert_customer(ck,nm,nk,bal) :
2.    m[][ordkey, sprior] +=
3.                      m_c[][ck, ordkey, sprior];
4.    m_l[][ordkey, sprior] +=
5.                    m_cl[][ck, ordkey, sprior];
6.    m_o[][ck] += 1;
7.
8.  on_insert_lineitem(ok,ep) :
9.    m[][ok, sprior] += ep *  m_l[][ok, sprior];
10.   m_c[][custkey, ok, sprior] +=
11.             ep *  m_cl[][custkey, ok, sprior];
12.   m_co[][ok] += ep;
13.
14. on_insert_order(ck,ok,sp) :
15.   m[][ok, sp]        += m_co[][ok] * m_o[][ck];
16.   m_l[][ok, sp]      +=  m_o[][ck];
17.   m_c[][ck, ok, sp]  += m_co[][ok];
18.   m_cl[][ck, ok, sp] += 1;
```

We briefly comment on one powerful transformation that is worth emphasizing in the above program, as seen on line 15. Notice that the right-hand side of the statement consists of two maps – all other statements are dependent on a single map. This line is an example of *factorization* applied as part of simplification. This statement is derived from the query q given at the start of the example, when considering an insertion to the `Order` relation with tuple $\langle$`@ck,@ok,@sp`$\rangle$:

```
q_o =  select   @ok,@sp,sum(l.extprice)
       from     Customers c, Lineitem l
       where    c.custkey = @ck
       and      l.ordkey = @ok;
```

The group-by clause of q can be eliminated since all group-by attributes are parameters. Note that the two relations `Customer`, and `Lineitem` have no join predicate, thus the query uses a cross product. By applying distributivity of the sum aggregate, we can separate (factorize) the above query into two scalar subqueries:

```
select @ok,@sp,
  sum(l.extprice)
from
  Customers c,
  Lineitem l
where c.custkey = @ck
and l.ordkey = @ok;
```

```
=>
```

```
select @ok,@sp,
((select sum(l.extprice)
  from   Lineitem l
  where  l.ordkey = @ok)
*
 (select sum(1)
  from   Customers c
  where  c.custkey = @ck));
```

DBToaster materializes the scalar subqueries above as m_{co} and m_o in its program, and in particular note that prior to factorization we had a single delta query of degree 2, and after factorization we have two delta queries of 1. The latter is clearly simpler and more efficient to maintain. Factorization can be cast as the generalized distributive law (GDL) [3] applied to query processing. GDL facilitates fast algorithms for many applications including belief propagation and message passing algorithms, and Viterbi's algorithm. With this analogy, we hope to leverage other techniques from this field, for example approximation techniques.

Transition program properties. For many queries, compilation yields *simple* code that has no joins and no nested loops, only single-level loops that perform probing as in hash joins. Simple code is beneficial for analysis and optimizations in machine compilation and code generation.

Transition programs leverage more space to trade off time by materializing delta queries. These space requirements are dependent on the active domain sizes of attributes, and often attributes do not have many distinct values, for example there are roughly 2800 unique stock ids on NASDAQ and NYSE. Additionally pruning duplicate maps during compilation facilitates much reuse of maps given recursion through all permutations of updates. Finally, there are numerous opportunities to vary space-time requirements for transitions: we need not materialize all higher-level deltas. For example we could maintain q with $m^{(i)}$, a materialized i-th level delta and perform more work during the update to evaluate $q^{(i-1)}, \ldots, q^{(1)}$. We could further amortize space by exploiting commonality across multiple queries, merging maps to service multiple delta queries.

Insights. Queries are closed under taking deltas, that is, a delta query is of the same language as the parent query. This allows for processing delta queries using classical relational engines in IVM. However, the aggressive compilation scheme presented above allows to innovate in the design of main-memory query processors. In the above example, we have materialized all deltas, thus the transition program consists of simple arithmetics on parameters and map lookups.

Our concept of higher-level deltas draws natural analogies to mathematics. Our framework, and the compilation algorithm described here – but restricted to a smaller class of queries without nested aggregates – was described and proven correct in [23]. In this framework, queries are based on polynomials in an algebraic structure – a *polynomial ring* – of generalized relational databases. This quite directly yields the two main properties that make recursive compilation feasible – that the query language is closed under taking deltas and that taking the delta of a query reduces its degree, assuring termination of recursive compilation. Unfortunately, the second property is not preserved if one extends the framework of [23] by nested aggregates. We describe be-

low how DBToaster handles this generalized scenario.

Discussion. To summarize, in contrast to today's IVM, DBToaster uses materialization of higher-level deltas for continuous query evaluation that is *as incremental as possible*. DBToaster is capable of handling a wide range of queries, including, as discussed next, nested queries. This has not been addressed in the IVM literature, and lets our technique cover complex, composed queries, where being as incremental as possible is highly advantageous.

3.2 Compilation Enhancements

We briefly discuss further compilation issues and optimizations beyond the fairly simple query seen in Figure 2.

Nested queries. We can compile transitions for nested queries, which has not been feasible in existing IVM techniques. In particular nested scalar subqueries used in predicates are problematic because taking deltas of such predicates does not result in simpler expressions. Our algorithm would not terminate if we did not handle this: we explicitly find simpler terms and recur on them. VWAP in Section 1 exemplifies a nested query.

Nested subqueries contain correlated attributes (e.g. price in VWAP) defined in an outer scope. We consider correlated attributes as parameters, or, internally in our framework, as binding patterns as seen in data integration. Nested queries induce *binding propagation*, similar to sideways information passing in Datalog. That is, we support the results of one query being used (or *propagated*) as the parameters of a correlated subquery, indicating an evaluation ordering. We transform queries to use minimal propagation, which performs additional aggregation of maps, over dimensions of the map key that are not propagated. For example a map $m[x, y, z]$ would be aggregated (*marginalized*) to $m'[x, y]$ if x, y were the only correlated attributes.

Rethinking query compilation with programming languages and compiler techniques. Our current compilation process involves implementing transition programs in a variety of target languages, including OCaml and C++. We currently rely on OCaml and C++ compilers to generate machine code, and observe that there are a wide variety of optimization techniques used by the programming languages (PL) and compiler communities that could be applied to query compilation. Compiling queries to machine code is not a novel technique, and has been applied since the days of System R [9]. However there have been many advances in source code optimization since then, as evidenced by several research projects aimed in this direction [4, 24].

We believe the advantage of incorporating methods from the PL and compiler communities directly into our compiler framework is that it facilitates whole-query optimizations, programmatic representation and manipulation of physical aspects of query plans such as pipelining and materialization (memoization), and opportunities to consider the interaction of query processing and storage layouts via data structure representations. Specifically, we have developed a small functional programming (FP) language as our abstraction of a physical query plan, unlike the operator-centric low-level physical plans found in modern database engines (e.g. specific join implementations, scans, sorting operators that make up LOLEPOPs in IBM's Starburst and DB2 [28], and similar concepts in Oracle, MS SQL Server amongst others).

The primary features of our FP language are its use of nested collections (such as sets, bags and lists) during query processing, its ability to perform structural recursion (SR) [8] optimizations on nested collections, and its support for long-lived persistent collections. Structural recursion transformations enable the elimination of intermediates when manipulating collections, and when combined with primitive operations on functions, such as function composition, yields the ability to adapt the granularity of data processing. Consider a join-aggregate query $\sum_{a*f}((R \bowtie S) \bowtie T)$ with schema $R(a, b)$, $S(c, d)$, $T(e, f)$, where the natural joins are Cartesian products. While such a query would not occur as a delta query in DBToaster (it would be factorized as discussed above), it suffices to serve as a toy example. Our functional representation is:

```
aggregate(fun < <t,u,v,w,x,y>, z>. (t*y)+z, 0,
  flatten(
    map(fun <w,x,y,z>.
          map(fun <e,f>.<w,x,y,z,e,f>, T),
        flatten(
          map(fun <a,b>.
                map(fun <c,d>.<a,b,c,d>, S),
              R)))))
```

Above, `fun` is a lambda form, which defines a function possibly with multiple arguments as indicated by the tuple notation. Next, `map`, and `aggregate` are the standard functional programming primitives that apply a function to each member of a collection, and fold or reduce a collection respectively. For example, we can use `map` to add a constant to every element of a list as:

```
map(fun x. x+5, [10;20;30;40]) => [15;25;35;45]
```

Similarly we can use aggregate to compute the sum of all elements of a list, with initial value 0 as:

```
aggregate(fun <x,y>. x+y, 0, [10;20;30;40]) => 100
```

The `flatten` primitive applies to a collection of collections (i.e. a nested collection), yielding a single-level collection. For example:

```
flatten([[1;2]; [3]; [4;5;6]]) = [1;2;3;4;5;6]
```

Our functional representation first joins relations R and S, before passing the temporary relation created to be joined with T, again yielding an intermediate, that is finally aggregated. Notice the intermediate flatten operations to yield a first normal form.

A standard implementation of a join as a binary operation forces the materialization of the intermediate result $R \bowtie_\theta S$. There are numerous scenarios where such materialization is undesirable, and has led to the development of multiway join algorithms such as XJoin [41] and MJoin [42]. With a few simple transformation steps, we can rewrite the above program to avoid this intermediate materialization as:

```
aggregate(fun < <t,u,v,w,x,y>, z>. (t*y)+z, 0,
  flatten(flatten(
    map(fun <a,b>. map(fun <c,d>. map(fun <e,f>.
      <a,b,c,d,e,f>, T), S), R))))
```

This is a three-way nested loops join $\bowtie_3$ (R, S, T) with no intermediate materialization of a two-way join, that can also be pipelined into the aggregation. In general, we can apply well-established folding [5], defunctionalization [13] and deforestation [27] techniques from functional programming, which when combined with data structures such as indexes

and hash tables, can yield a rich space of evaluation strategies that vary in their pipelining, materialization, ordering, nesting structure and vectorization characteristics.

The last item, nesting structure and vectorization, refers to concepts from nested relational algebra [38], whereby query processing need not occur in terms of first normal form relations. Our programs can use non-first normal forms internally, and can apply compression and vectorized processing over the nested relation attributes, much in the vein of column-oriented processing. Furthermore, we can directly represent the aforementioned data structures, such as indexes and DBToaster maps, in our language, yielding a unified approach to representing both query processing and storage layout. We know of no existing framework capable of such a rich representation, and are excited by the potential to apply program transformations to jointly optimize query processing and storage.

4. MANAGING STORAGE IN DBTOASTER

We now examine two components of DBToaster's solution to storage in a DDMS: (1) The DBToaster compiler produces data structures designed specifically for the compiled DDMS' target query workload. (2) By analyzing the patterns with which data is accessed, DBToaster constructs a data layout strategy (for pages on a disk, servers in a cluster, etc.) that limits IO overhead.

Data structures. DBToaster uses multi-key (i.e., multi-dimensional) maps to represent materialized views. The generated code performs lookups of slices of the maps, using partial keys, fixing some dimensions and iterating over the others. Recall the right-hand side of line 2 in the code listing in Section 3. This is a partial lookup on map m_c with only `ck` defined by the trigger arguments. In addition to exact and partial lookups, DBToaster maps support range lookups by inequality predicates.

In their simplest form, out-of-core maps are implemented by a simple relational-style key-value store with secondary indices [30]. Inequality predicates, and aggregations including such predicates, are implemented efficiently using maps that store cumulative sums [19]. Maps can apply compression techniques to address frequently repeating data. DBToaster customizes the data structures backing each materialized view based on statement-level information on accesses, applying static compile-time techniques to construct specialized data structures.

With substantial specialization of data structures as part of compiling transitions, DBToaster is free to consider a range of runtime issues in data structure tuning and adaptation, including how to best perform fine-grained operations such as incremental and partial indexing [39]. The key challenge to be addressed is how to provide data structures with a low practical update cost (avoiding expensive index rebalancing and hash-table rebucketing) while gradually ensuring the lookup requirements of our data structures are retained over time, amortizing data structure construction with continuous query execution.

Partitioning and co-clustering by data flow analysis. Database partitioning and co-clustering decisions are traditionally made based on a combination of (a) workload statistics, (b) information on schema and integrity constraints (such as key-foreign key constraints, a popular basis for co-clustering decisions), and (c) a body of expert insights into how databases execute queries. Ideally, such decisions should be based on a combination of (a), (b), and a *data flow analysis* of the system's query execution code, instantiated with the query plan, or view maintenance code. In classical DBMS however, this is too difficult to be practical.

Fortunately, data flow analysis turns out to be feasible for compiled DDMS transition programs: in fact, it is rather easy. A transition program statement reads from several maps and writes to one, prescribing dependencies between those maps occurring on the right-hand side of the statement (reading), and the one on the left-hand side (writing). As pointed out in Section 3, transition program statements admit a perfectly data-parallel implementation: consequently, a statement never imposes a dependency between two items of the same map and any horizontal partitioning across the involved maps map keeps updates strictly local. Using these data flow dependencies, partitioning and co-clustering decisions can be made by solving a straightforward min-cut style optimization problem.

5. DBTOASTER IN THE CLOUD

Scaling up a DDMS requires not only storing progressively more data, but also a dramatic increase in computing resources. As alluded to in Section 4, DDMS and their corresponding transition programs are amenable to having their data distributed across a cluster: (1) The only data structures used by transition programs are maps, which are amenable to horizontal partitioning. (2) At the granularity of a single update, iterative computations are completely data-parallel.

Update Processing Consistency and Isolation. In DBToaster, transition functions are created under the assumption that they are operating on a *consistent* snapshot of the DDMS's state. The entire sequence of statements composing the trigger function must be executed atomically, to ensure that each statement operates on maps resulting from fully processing the update stream prior to the update that fired the trigger. Thus the effects of updates should be fully isolated from each other. Similar issues and requirements have been raised before in the single-site context of view maintenance with the "state bug" [10].

Our requirement of processing updates in such an order is conservative, indeed we could apply standard serializable order concurrency control here to simultaneously process updates that do not interact with each other. The underlying goal is to develop simple techniques that avoid heavyweight locking and synchronization of entries in massively horizontally partitioned maps. We start with a conservative goal to focus on lightweight protocols. Ensuring this atomicity property is the first of the two core challenges that we encountered while constructing a distributed DDMS runtime.

Distributed Execution. Each update in our distributed DDMS runtime design employs three classes of actor:

- *source nodes*: Nodes hosting maps read by the update's trigger function (maps appearing on the right-hand side of the function's statements).
- *computation nodes*: The nodes where statements are evaluated.
- *destination nodes*: Nodes hosting maps written to by the update's trigger function (maps appearing on the left-hand side of the function's statements).

Note that these actors are logical entities; it is not necessary (and in fact, typically detrimental) for the actors to be on separate physical nodes within the cluster. Introducing a distinction between the different tasks involved in update processing allows us to better understand the tradeoffs involved in the second core challenge: selecting an effective partitioning scheme that intelligently determines placements of logical entities in order to best utilize plentiful hardware to handle a large update stream and DDMS state.

5.1 Execution Models

We first address the issue of atomicity by providing two execution models: (1) A protocol that provides a serial execution environment for transition programs, and (2) An eventual consistency protocol that provides the illusion of serial execution.

Serial Execution. The most straightforward way of achieving atomicity is to ensure serial trigger function execution. However, requiring all nodes in the cluster to block on a barrier after every update is not a scalable approach. A similar effect can be achieved more efficiently by using fine-grained barriers, where each update is processed by first notifying all of the update's destination nodes of an impending write. Reads at the update's source nodes are blocked while writes from prior updates are pending.

Serial execution requires a global ordering of updates as they arrive at the DDMS. Techniques to achieve this include:

- Updates arrive only from a single producer (e.g., the cluster is maintaining a data warehouse that mirrors a single OLTP database).
- A central coordinator generates a global ordering (as in [32]).
- A distributed consensus protocol generates a global ordering (as in [22]).
- A deterministic scheme produces a global ordering. For example, each update producer generates timestamps locally and identical timestamps in a global view are settled with a deterministic tiebreaker like the producer's IP address.

We also need a mechanism to provide consistent delivery of updates from multiple producers. Before completing a read, source nodes must not only ensure that all prior pending writes have been completed, but also that all notifications for prior updates have been received. A simple solution is to channel all updates through a single server. This has the advantage of also providing a global ordering over all updates. However this solution creates a scalability bottleneck. Alternative solutions like broadcasting updates or periodic commits are possible, but introduce considerable synchronization overheads.

Speculative Execution with Deltas. As an alternative, we can favor the use of speculative and optimistic processing techniques in designing our distributed execution protocol. The key insight here is that our computation is based on incremental processing, thereby unlike standard usage of speculative execution, any work done speculatively need not be thrown away entirely, rather any work done can be revised through increments or deltas, to the final desired outcome.

In particular, a node can optimistically perform reads immediately (or at least, blocking only on pending write operations which the node is already aware of, and not the vague possibility of potential future write operations). Although avoiding blocking on potential future writes eliminates significant synchronization overheads, out-of-order updates can cause the atomicity and desired ordering properties of trigger execution to be lost. We favor this point in the design space since out-of-order events are expected to occur infrequently. Furthermore, such events are likely to interfere with only a handful of prior updates, for example a write on one map entry followed by an out-of-order read on a different entry in the same map do not cause a problem. Finally, because write operations are limited to additive deltas, there is a clear mechanism for composing out-of-order writes.

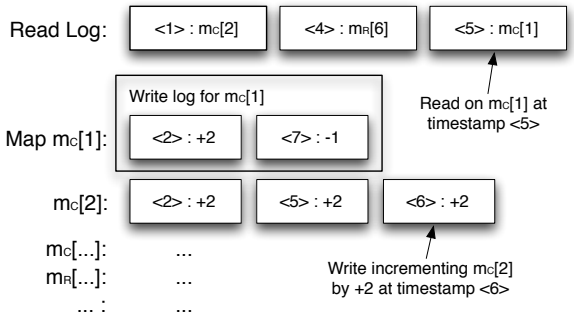


Figure 3: Supplemental data structures used to facilitate speculative execution in a distributed DDMS.

Out-of-Order Processing with Deltas as Revisions. Two types of out-of-order operations can occur in the speculative execution model: write-before-read, and read-before-write. We supplement maps with two additional data structures capturing timestamp information for operations, as illustrated in Figure 3: (1) Source nodes maintain a log of all read operations. (2) Destination nodes save all write operations independently; map entries are saved as logs rather than summed values. Each operation is tagged with and sorted by the effecting update's timestamp `<t>`.

In the case of an out-of-order read operation (i.e., one that arrives after a write operation that logically precedes it), the write log makes it possible to reconstruct the state of the map at an earlier point in time. For example, given the initial state in Figure 3, an update that requires a read on entry m_C`[2]` arrives with timestamp `<3>`. The value sent to the computation nodes is not the latest value of the entry (m_C`[2]` $= 6$ for all timestamps after `<6>`), but rather the sum of all values with lower timestamps (m_C`[2]` $= 2$ for timestamps `<3>`,`<4>`, and `<5>`).

In the case of an out-of-order write operation, the read log allows us to send a *revising update* to each computation node affected by the write. For example, given the initial state in Figure 3, an update that requires a write on entry m_R`[6]` arrives with timestamp `<3>`. The value will be written as normal (i.e., inserted into the write log for m_R`[6]`, in sorted timestamp order). Additionally, because the read log shows a read on the same entry with a later timestamp, a corrective update will be sent to the computation node(s) to which the entries were originally sent to.

Both data structures grow over time. To prevent unbounded memory usage, it is necessary to periodically truncate, or garbage collect the entries in each. This in turn, requires the runtime to periodically identify a cutoff point,

the "last" update for which there are no operations pending within the cluster. The read history is truncated at this point, and all writes before this point are coalesced into a single entry. Though this process is slow, it does not interfere with any node's normal operations, and can be performed infrequently, for example once every few seconds.

Hybrid Consistency. While the speculative execution model and its eventually consistent results are advantageous from a performance and scalability perspective, there may not be a point at which the state of all maps in the system corresponds to a consistent snapshot of our transition programs evaluated over any prefix of the update stream. That is, there is no guarantee that the system has actually converged to its eventually consistent state in the presence of a highly dynamic update stream. However, a side effect of the garbage collection process is that each garbage collection run, in effect generates a consistent snapshot of the system. As in other eventual consistency systems [14], this approach offers a hybrid consistency model, specifically the same infrastructure produces both low-latency eventually consistent results, as well as higher-latency consistent snapshots.

5.2 Partitioning Schemes

The second challenge associated with distributing a transition program across the cluster is the distribution of logical nodes (source, computation, and destination) across physical hardware in the cluster. In addition to more complex, min-cut based partitioning schemes for the data, DBToaster considers two simple partitioning heuristics for distributing computation: (1) data shipping: evaluate program statements where their results will be stored, at destination nodes, or (2) program shipping: evaluate program statements where their input maps are stored, at source nodes.

Destination-Computation. Given the one-to-one correspondence between computation nodes and destination nodes, the simplest partitioning scheme is to perform computations where the data will be stored – that is, the destination and computation nodes are co-located. As part of update evaluation, each source node transmits all relevant map entries to the destination node. Upon arrival, the destination node evaluates the statement and stores the result.

Source-Computation. Though simple, transmitting every relevant map entry with every update can be wasteful, especially if the input map entries don't change frequently. An alternative approach is to co-locate all of the source nodes and the computation node. When evaluating an update, the computation can be performed instantaneously, and the only overhead is transmitting the result(s) to the destination node(s). This is particularly effective in queries where update effects are small (e.g., queries consisting mostly of equijoins on key columns).

However, this approach introduces an additional complication. It is typically not possible to generate a partitioning of the data that ensures that for each statement in a trigger program, all the source nodes will be co-located. In order to achieve a partitioning, data must be replicated; each map is stored on multiple physical nodes. While replication is typically a desirable characteristic, storage-constrained infrastructures may need to use complex partitioning schemes.

6. DISCUSSION AND CONCLUSIONS

We have proposed Dynamic Data Management Systems, arguing for the need for a class of systems optimized for keeping SQL aggregate views highly available and fresh under high update rates. We have sketched some of the main research challenges in making DDMS a reality, and have outlined key design decisions and first results and insights that make us confident that the vision of DDMS can be realized.

We are currently developing a DDMS, DBToaster, in a collaboration between EPFL and Johns Hopkins, which was started while the authors worked at Cornell. So far, we have developed an initial version of a transition compiler which implements the recursive IVM technique sketched in Section 3. The feasibility of keeping views fresh through hundreds of updates per second using this approach in the context of algorithmic trading was recently demonstrated using an early DBToaster prototype [2]. An initial account of the foundations and theory of recursive IVM was given in [23]. Apart from substantially improving the compiler, we are now working on the actual DBToaster DDMS. We follow the strategy of developing two branches, one a main-memory, moderate state-size, single-core ultra-high view refresh rate system for applications such as algorithmic trading and the other a cloud-based, persistent, eventual-consistency system for very large scale interactive data analysis. We will merge these two branches once the individual technical problems have been solved. Further details and an update stream on the project can be found on our website at `http://www.dbtoaster.org`.

Acknowledgments

This project was supported by the US National Science Foundation under grant IIS-0911036. Any opinions, findings, conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of NSF. We thank the Cornell Database Group for their feedback throughout the development of the DBToaster project.

7. REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[2] Y. Ahmad and C. Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2), 2009.

[3] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2), 2000.

[4] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, 2010.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4), 1994.

[6] D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *CAV*, pages 1–18, 2010.

[7] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR&DB integration. In *CIDR*, 2007.

[8] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1), 1995.

[9] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for repetitive transactions and ad hoc queries in System R. *ACM Trans. Database Syst.*, 6, 1981.

[10] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, 2006.

[12] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2003.

[13] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP*, 2001.

[14] D.B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[15] D. Deutch, C. Koch, and T. Milo. On probabilistic fixpoint and Markov chain query languages. In *PODS*, 2010.

[16] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM TODS*, 35(1), 2010.

[17] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.

[18] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: database-supported program execution. In *SIGMOD*, 2009.

[19] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, 1997.

[20] R. Iati. The real story of trading software espionage. AdvancedTrading.com, July 2009.

[21] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik. Towards a streaming SQL standard. *PVLDB*, 1(2), 2008.

[22] F. P. Junqueira and B. C. Reed. The life and times of a zookeeper. In *PODC*, New York, NY, USA, 2009.

[23] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[24] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[25] L. Brenna et al. Cayuga: a high-performance event processing engine. In *SIGMOD*, 2007.

[26] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, 1988.

[27] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Functional Programming*, 1992.

[28] J. McPherson and H. Pirahesh. An overview of extensibility in starburst. *IEEE Data Eng. Bull.*, 10(2), 1987.

[29] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and xml in the .net framework. In *SIGMOD*, 2006.

[30] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX*, 1999.

[31] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.

[32] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[33] R. G. Bello et al. Materialized views in Oracle. In *VLDB*, 1998.

[34] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[35] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms & cost analysis. *ACM TODS*, 16(3), 1991.

[36] S. Ceri et al. Practical applications of triggers and constraints: Success and lingering issues. In *VLDB*, 2000.

[37] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[38] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2), 1986.

[39] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4), 1989.

[40] G. J. Sussman and G. L. S. Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4), 1998.

[41] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2), 2000.

[42] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

[43] A. von Bechtoldsheim. Scalable networking for cloud datacenters. Invited Talk, EPFL, September 2010.

[44] W. M. White, M. Riedewald, J. Gehrke, and A. J. Demers. What is "next" in event processing? In *PODS*, 2007.

[45] M. L. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *PVLDB*, 3(1), 2010.

[46] Y. Fu et al. AJAX-based report pages as incrementally rendered views. In *SIGMOD*, 2010.

[47] J. Zhou, P.-Å. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, 2007.

[48] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *ICDE*, 2007.

SciBORQ: Scientific data management with Bounds On Runtime and Quality

Lefteris Sidirourgos
CWI
Amsterdam, the Netherlands
lsidir@cwi.nl

Martin Kersten
CWI
Amsterdam, the Netherlands
mk@cwi.nl

Peter Boncz
CWI
Amsterdam, the Netherlands
boncz@cwi.nl

ABSTRACT

Data warehouses underlying virtual observatories stress the capabilities of database management systems in many ways. They are filled, on a daily basis, with large amounts of factual information derived from intensive data scrubbing and computational feature extraction pipelines. The predominant data processing techniques focus on parallel loads and map-reduce feature extraction algorithms. Querying these huge databases require a sizable computing cluster, while ideally the initial investigation should run interactively, using as few resources as possible.

In this paper, we explore a different route, one based on the observation that at any given time only a fraction of the data is of primary value for a specific task. This fraction becomes the focus of scientific reflection through an iterative process of ad-hoc query refinement. Steering through data to facilitate scientific discovery demands guarantees for the query execution time. In addition, strict bounds on errors are required to satisfy the demands of scientific use, such that query results can be used to test hypotheses reliably.

We propose SciBORQ, a framework for scientific data exploration that gives precise control over runtime and quality of query answering. We present novel techniques to derive multiple interesting data samples, called *impressions*. An *impression* is selected such that the statistical error of a query answer remains low, while the result can be computed within strict time bounds. *Impressions* differ from previous sampling approaches in their *bias* towards the focal point of the scientific data exploration, their *multi-layer* design, and their *adaptiveness* to shifting query workloads. The ultimate goal is a complete system for scientific data exploration and discovery, capable of producing quality answers with strict error bounds in pre-defined time frames.

1. INTRODUCTION

Scientific instruments produce huge amounts of information which is stored in large data warehouses. Examples are virtual observatories populated with astronomical data, or the Grid, a computer cluster spanning the globe with experimental data originating from the Large Hadron Collider at CERN. The data produced is so large that in many cases a decade of intense exploration by the scientists passes by before new observations are obtained and safe conclusions are drawn. The predominant data processing techniques focus on massivly parallel loads and distributed processing on a computer cluster. Although these approaches allow efficient execution of complicated and computationally intensive workflows, they do not provide interactive and low-cost means for the scientists to make an initial exploration over the daily produced data. The demand for data intensive scientific discovery led Jim Gray to call the community to arms to face the challenge of the "Fourth Paradigm" [11]. Facing this challenge calls for a database architecture exhibiting features different from contemporary ones.

A significant portion of the processing time goes into loading data into the science data warehouse and to prepare it for fast retrieval. The daily ingest may involve data sizes that are already hard to manage. Indexing may take an exorbitant amount of time, otherwise, massive data parallel processing is needed later on. Even a raw scan is hindered by the sequential bandwidth required. Our hypothesis is that in many real-life situations the scientist is initially satisfied with a properly chosen database sample as a starting point for determining a query scenario. This scenario, once proven correct and relevant, can be run in depth against all data overnight. The key challenge is to determine what constitutes a good set of sampled data, such that the interests of the scientist are met and the computational tasks run efficiently, thus providing interactive query performance. Traditional approximate query answering and online aggregation methods do not satisfy the requirement of complete control over both resource consumption and query result error bounds.

In this paper we describe SciBORQ [1], a novel architecture that extracts multiple samples of a science database, called *impressions* hereafter, to facilitate data exploration with guarantees on execution time and tight error bounds. The approach taken generalises the *sampling* techniques originally designed to maintain synopsis and histograms for query optimisation. Contrary to existing work, *impressions* are large samples *biased* towards the scientist's interest as captured by taking note of the query workload. SciBORQ constantly *adapts* towards the shifting focal points of real time data exploration. *Adaptive biased sampling* is more suitable under the observation that given a limitation on size, it is better to pick more tuples from the areas of interest so as to minimise the error bounds. New challenges emerge, such as providing correct estimators, satisfactory error bounds, and execution time guarantees.

Unlike synopsis and histograms, which are traditionally used for approximate query answering, the size of an *impression* may be many gigabytes rather than just kilobytes or megabytes. Query processing is designed such that a query may be evaluated against multiple impressions, according to the specific user demands on er-

[1] pronounced as *cyborg*

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

ror and time bounds. Therefore, multiple *impressions* of different size and focus are derived. Depending on the policy chosen, some scientists would be keen to keep the latest observations in their samples, while others may only be interested in events close to a point of interest. Others may be interested in the outliers, i.e., peaks or troughs of the data instead of average values. Finally, for practical data exploration, it is imperative to control the statistical errors that might occur when a database query is executed, or bound the processing time to an acceptable limit, for example, "*give me the most representative result you can obtain within 5 minutes*".

The key features of SciBORQ can be summarised as follows:

- SciBORQ consists of *impressions*, which are created and updated incrementally during parallel database loads, such that a scientist's interest captured by an *impression* is satisfied.
- *Impressions* adaptively reflect the focal point of scientific exploration, which is derived from the query workload.
- *Bounded query processing* is facilitated by recursively defined *impressions* with strict control over their response time, disk space, and statistical quality, leading to a *multi-layer* data exploration framework.

The research opportunities under such an architecture are promising. *Biased adaptive multi-layered samples* are an entirely new concept, introducing new areas for research in database theory and system design. Moreover, the specifics of sampling over a read-optimised columnar architecture have not been studied in detail yet – leaving ample space for exploration and rethinking of already established sampling techniques. Bounded query answering calls for developing a new query processing framework that can keep errors under control by resorting to using more detailed impressions, or in the extreme case, the base data. Finally, although there is an articulated desire from the scientific community to provide database engines with control over the execution time [22], no significant steps have been done towards the realisation of such a system.

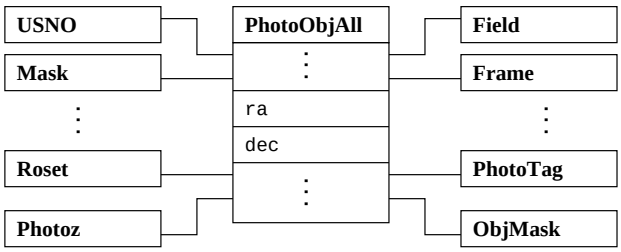
The rest of the paper is organised as follows. Section 2 presents one of the scientific data warehouses that motivates our work. Section 3 presents the design of SciBORQ. Section 4 details the concept of adaptive and biased sampling. Section 5 presents related work, followed by Section 6 with conclusions and future work.

2. SCIENTIFIC DATA WAREHOUSES

The proposed multi-layer query processing framework is targeted towards an ongoing astronomy applications, the Sloan Digital Sky Survey SkyServer. The SciBORQ implementation is designed to work on top of MonetDB [17], a modern column-store database system with a proven track record in various fields [12, 13, 16]. MonetDB is already integrated with the aforementioned application as the underlying data management system.

2.1 Sloan Digital Sky Server

The Sloan Digital Sky Server realisation in SkyServer [2] is a well-known and complex science data warehouse. Its schema encompasses several tens of relational tables. Figure 1 shows a summarised view of the schema. The main fact table `PhotoObjAll` contains hundreds of columns and several billion tuples. Each tuple contains information about an astronomical image. Attribute `ra` refers to the right ascension, and `dec` to the declination of the image in the sky. More information is incorporated by joining the foreign key attributes of the main fact table to the dimension tables. In addition, the SkyServer schema contains tens of views and functions to facilitate data exploration. A fully functional implementation of this 4TB database is available for MonetDB. The publicly accessible query logs provide a basis to derive areas of interest. A large percentage of the queries have the form shown in the lower part of Figure 1. Table `Galaxy` is a view of `PhotoObjAll` with many foreign key joins. This view presents the *galaxy* information according to the astronomers' desire. The function `fGetNearbyObjEq` returns all objects found in a nearby area specified by `ra=185` and `dec=0`. The scientists' interest can be satisfied, even if only the data around the coordinates `ra` and `dec` are available, and not the entire data set. The area described by the query predicate is the focal point of exploration. Often this focal point is limited to a small part of the sky. Queries can run anywhere from a few seconds on a large cluster, to tens of minutes on a single machine.




```
select *
from Galaxy as G,
dbo.fGetNearbyObjEq(185,0,3) as N
where G.objID = N.objID
```

Figure 1: SkyServer Schema and Query

The SkyServer application is prototypical for emerging projects, such as Pann-Stars and LSST. The system is used by around 2000 astronomers worldwide to support and drive their research. The majority of users, however, consists of amateur astronomers challenging the system with a large and complex query load.

3. SciBORQ ESSENTIALS

The key to multi-layer query processing is to extract samples from the database, the *impressions*, such that bounded query processing functionality is precisely controlled. Impressions are of different size, ranging from a few kilobytes to many gigabytes. Depending on their size, an impression fits either in the CPU cache, or the main memory of a workstation, or resides on the disk of a laptop or even a cluster. Therefore, this flexibility has a direct impact on the execution time of a query, the number of results produced, and the answer quality. Impressions bear commonalities with data synopsis and histograms but their purpose, functionality, and applicability go beyond that. In this section we sketch the landscape of the SciBORQ system.

3.1 System Parameters

Size. Impressions have different sizes with different degrees of detail. The memory footprint of an impression is directly proportional to the error bounds and the processing time that can be promised. The larger the impression, the longer the processing time and the smaller the error bounds. The user is able to define the desired size of an impression to serve her needs.

Focal point. An impression gathers data according to a sampling strategy. However, the sampling need not be from the entire database, but can be from specific areas of interest. The focal point of an impression is defined to be exactly this area of interest. The

[2] http://www.sdss.org

predicates and the join conditions of the queries in a workload determine what is important for the scientist and what not. For example, in the SkyServer paradigm, by requesting objects of the galaxy with the `fGetNearbyObjEq` function, effectively the scientist is defining one of the focal points for an impression.

Layers. SciBORQ is a multi-layer hierarchical and parallel collection of impressions. Impressions are defined to serve different purposes and needs of the application user. In traditional systems, there is one (typically small) synopsis of the data (i.e., sample, catalogue statistics) used by the query optimiser or for approximate query answering. However, in SciBORQ multiple impressions of different sizes and focal points are constructed. Each less detailed impression is derived from a previous more detailed one. In such a derivation, the focal point of the larger impression is inherited by the smaller, but many such hierarchies of impressions exist. If the error bounds during query execution are not met, the process continues on a larger impression of the same hierarchy. Moreover, smaller impressions on higher layers are more efficient to maintain since they only touch the data of the impression one layer below, and not the entire base. This is important, since small impressions need fast reflexes to efficiently adapt to query workload shifts.

Correlations. Impressions do not contain just a single attribute or relation, but may span the entire database logical schema. Each one of them may reflect the total of the base tables, or since SciBORQ is designed for read optimised column stores, may contain a subset of the attributes of a table. If the need rises, more columns can be added. Past work [3, 4, 18, 21] demonstrates how join attributes across relations are achieved with uniform sampling, and it can be adjusted to our case, too. This way, the correlations between join attributes are maintained, leading to more precise query results.

Adaptive. An impression constantly adapts to the focal point of the scientist's exploration, such that it contains more data from the areas of interest. To achieve this objective, there are two phases where an impression has the opportunity to re-adjust its focus: as a side-effect of query processing and, alternatively, by triggering impression maintenance on subsequent incremental loads. SciBORQ recognises tuples that are potentially interesting for the workload that has been observed up until now, thus increasing their chance of being part of the corresponding impression. This strategy ensures better resolution around the focal points.

3.2 Bounded Query Processing

Quality of results. An important feature of the SciBORQ design is the quality guarantees given for the query results. Any scientific exploration, no matter how generic, is useful only if strong error bounds are provided. Although traditional sampling techniques provide confidence levels on the results, error bounds will deteriorate due to correlations and complicated query plans. If a scientist is prepared to accept only a specific upper limit on the error, he will be left unsatisfied. A new query execution engine is needed that can dynamically keep the error bounds under control. In SciBORQ, if the error bound requested is not met during execution, the query evaluation moves to an impression on a lower level, with a higher level of detail, to confine the error margin. Ultimately, this can lead to the base columns for a zero error margin. The implementation of such functionality is feasible because of the special runtime optimisation capabilities of a system such as MonetDB that materialises intermediate results and provides the hooks to dynamically change the query plans [15]. In addition, since query processing is column oriented, some operators with low statistical confidence can run on a larger impression of the same hierarchy, while other operators can ran on a smaller one.

```
populate the sample smp with the first n tuples;
cnt := n;
while (tpl := block_until_next_tuple())
      cnt++;
      rnd := floor(cnt*random());
      if (rnd < n)
         smp[rnd] := tpl;
      end
end
```

Figure 2: Reservoir algorithm R

Execution time. In today's systems, the amount of results that a query produces can only be limited by a count barrier, i.e., LIMIT clause in SQL. Indirectly, the query execution time can be controlled likewise. Its implementation relies on "cutting" the execution pipeline when enough tuples have been produced or the predefined timeout is triggered. The main problem with this approach is that the *first* N results are returned, where *first* is defined arbitrarily by the order in which the data is processed. This order can be either user defined (e.g., an ascending numerical order), or the order in which the data was appended to the relation, or, finally, defined by an index for fast retrieval. In all cases, such a cut does not necessarily produce representative results for the entire data population, but merely the lucky N first tuples. Moreover, in the presence of blocking operators, such as *'sort'*, *'group by'*, etc., all data has to be read to produce the correct answer, and thus the pipeline cannot be cut. Query processing in SciBORQ is much different in that respect. The parameters of impressions are defined and maintained during updates, such that SciBORQ always guarantees an upper limit on time execution while producing results that are sampled from the entire database. In such an architecture the equivalent query with a `LIMIT 100` clause will not return the first 100 results, but the 100 results satisfying the impression. In the SkyServer example, instead of finding all objects near an area of the galaxy by evaluating the function `fGetNearbyObjEq` against the entire `PhotoObjAll` fact table, and then returning only a few results, the function is evaluated against an impression. If the number of results cannot be obtained from that impression, query processing may continue to a lower level that contains more sampled tuples from `PhotoObjAll`.

3.3 Impressions Construction

Impressions are deployed either as part of a database loading step or extracted from an existing database. In the first case, they are constructed with little overhead during the load phase, without the need to visit the base tables after the data is stored. The construction algorithms reside in the load process, considering each tuple as it is being loaded, much like a stream, and deciding if it should be part of an impression or not. Because daily ingests of new data are common in scientific data warehouses, the algorithms for creating an impression also support incremental updates.

The incremental construction of impressions follow the *reservoir algorithms* paradigm [24]. Reservoir algorithms have a) a fixed capacity of tuples that can fit in the sample, b) process the data sequentially, and c) each tuple has the same probability of being part of the sample. The size of the sample is kept constant by throwing out a random tuple to make room for a newly arrived one. Figure 2 outlines the general reservoir algorithm for maintaining a sample of size n. The decision to include or not a tuple in the sample is equal to flipping a coin with probability of acceptance $\frac{n}{cnt+1}$, where cnt is the number of tuples seen so far. Our algorithms stress the definition of reservoir algorithms, since in SciBORQ tuples are not chosen uniformly.

```
populate the sample smp with the first n tuples;
while (tpl := block_until_next_tuple())
      rnd := random();
      if ((D*rnd) < k)
         smp[floor(n*rnd)] := tpl;
      end
end
```

Figure 3: Last Seen Impression construction

Scientific observations have a strong temporal component. It is often more important to retain recent tuples than ones that have been investigated several times already. This leads to a *Last Seen* focused impression, where tuples that were recently added have a greater probability of being retained. To achieve this, instead of picking a tuple with probability $\frac{n}{cnt+1}$, we use the fixed probability $\frac{k}{D}$, where D can be tuned to be close to the expected daily ingest of new tuples, and $k = n$ if only new tuples are desired, or $k < n$ for a ratio of $\frac{k}{n}$ new tuples in the sample. In such a strategy, older tuples have a bigger chance of being thrown out from the reservoir. Figure 3 outlines this algorithm. The *Last Seen* approach is useful in cases where observations have a timestamp which is used in query predicates.

The second strategy for determining the scientist's interest is based on a more complex infrastructure of query logging. Every query ran against the complete database touches a subset of the base tables that are relevant to the data exploration. An approach would be to keep this set as an impression. The MonetDB *recycler* component already facilitates this functionality [13]. Here we seek an algorithm such that the probability of keeping a tuple is proportional to the *distance* of the values of that tuple from the values requested by the query workload. For each predicate of a query, the requested values are logged in histograms. These histograms do not contain the entire value space of an attribute, but only a portion. Given the workload knowledge, for each ingested tuple a weight is calculated and used to *bias* the sample towards the tuples with higher weight. In the next section we present the essentials of *biased sampling* and how the weight is calculated.

Finally, we can incorporate biased sample construction across many-to-many joins and foreign key joins by following each join path [3], or by using weighted sampling [4]. However, due to the special nature of impressions (i.e., incremental and adaptive biased sampling), these traditional sampling techniques have to be adapted to *wait* for the joining tuples to arrive during subsequent loads.

4. BIASED SAMPLING

Biased sampling is achieved by assigning weights to tuples such that those that belong to areas of past interest have a higher probability to be part of an impression than other, irrelevant ones. Intuitively, the upside is that queries that target the area of interest have tighter error bounds. The downside is that the confidence of queries that span widely outside of these areas is lower. Assigning weights to the probability of picking an item leads to a *non-central hypergeometric distribution*. Specifically, our setting is described by the *Fisher's non-central hypergeometric distribution* [6]. These mathematical tools provide the theory to calculate the variance, the mean, and the support function of the biased sample.

Biased sampling is steered by the observed interest in the data. This is achieved by first identifying the attributes of the data that contain relevant scientific observation values rather than annotations or metadata. In the SkyServer setting of Figure 1, these attributes, for the main fact table `PhotoObjAll`, are for example `ra` and `dec`, which give the position of the observed objects in the sky. They appear as parameters of the `fGetNearbyObjEq` function. For many of the queries in the workload, these attributes are part of the `WHERE` clause. Given a query workload – which is defined over a period of time or over a predefined number of queries – the *predicate set* is the set of all values of the interesting attributes that are requested by the queries. During incremental load of data into the `PhotoObjAll` fact table, tuples are sampled with a bias to the areas of the sky that previously appeared in the predicate set.

```
struct histo_stats{int c=0;
                   float m=0;
                  } hs[β];
N = 0;
while (v := block_until_next_value())
      N++;
      i := floor((v-min)/w);
      hs[i].c++;
      hs[i].m=(hs[i].m×(hs[i].c-1)+v)/hs[i].c;
end
```

Figure 5: Histogram maintenance over the predicate set

The values in the predicate set are regarded as points that *suggest* the entire distribution of values of interest. A *kernel density estimator (kde)* is used to estimate this interest. Kernel density estimators have been used to approximate the distribution of a sampled space [20]. They are smoother than histograms because they avoid rounding errors, and there is no dependency on the endpoints or the width of the bins of a histogram. Moreover, since they are continuous, and not discrete, they give a better view of the neighbour area of the observed values. Assume a set of N data points $x_1, \ldots, x_N$ as they appear in the predicate set of a query workload. The kernel density estimator estimates the expected total workload and is given by the function:

$$\hat{f}(x) = N^{-1} \sum_{i=1}^{N} K_h(x - x_i)$$

where $K_h(\cdot) = h^{-1}K(\cdot/h)$ where K is a kernel function and h the bandwidth. A common choice of K is the standard normal (Gaussian) distribution $\phi(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}$. Function $\hat{f}$ is an estimator of the density function f of requested values, given N data points.

Figure 4 depicts two equi-width histograms that correspond to the distribution of 400 values as observed in the predicate set for attributes `ra` and `dec`. An important parameter is the choice of h, called the *bandwidth* of the kde. The red lines of Figure 4 show the density function estimation of the values in the histograms, as approximated by function $\hat{f}$ with a carefully chosen bandwidth. Notice that a large h will *oversmooth* the distribution (green lines in Figure 4), while a small h will *undersmooth* (blue lines in Figure 4). Choosing the correct approximation for the bandwidth h is hard and has been an area of intense research [14]. Moreover, computing $\hat{f}$ for a new value x involves re-iterating over all observed values $x_1, \ldots, x_N$. This implies that for every newly ingested tuple t_{new} the computation of $\hat{f}(t_{new})$ involves reading all N previously observed values of the predicate set. We adjust the kde to our setting to overcome these shortcomings as follows.

The first step is to maintain statistical information of equi-width histograms for the attributes of interest to the scientific exploration. These values are exactly the ones requested by the queries of the workload and not the entire value domain. For the previous example of SkyServer and attributes `ra` and `dec`, we maintain statistics for two histograms [3]. These histograms are different from the

[3]multi-dimensional histograms are more attractive, but for simplicity of the example we use two distinct histograms. Alternative ap-

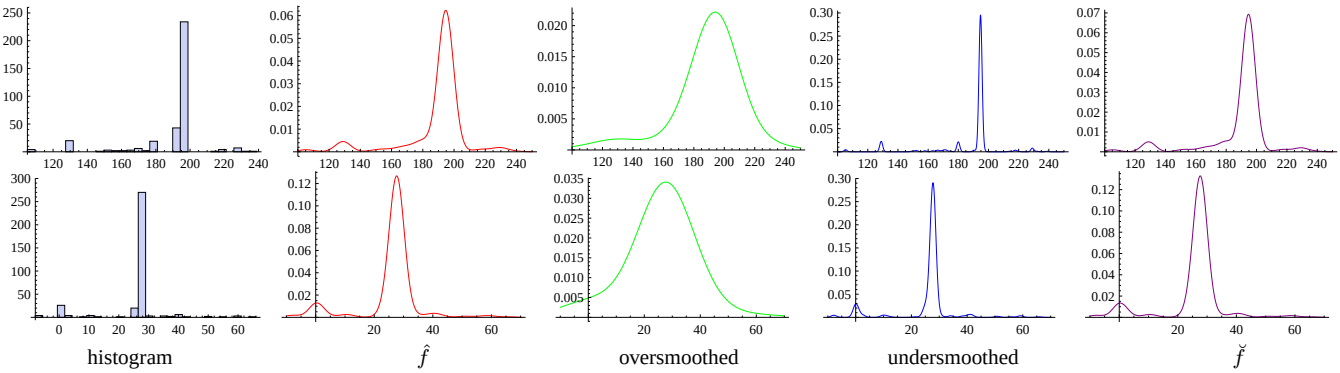


Figure 4: 1st row is for predicate `'ra'` and 2nd row for `'dec'`

ones shown in Figure 4, since they are not fully materialised as the figure suggests. Only the statistical aspects of the histograms are needed: the number of values that fall in a specific bin and their mean value. More specifically, the domain of each attribute is divided into β equal-width bins. The width is denoted by w. For each bin $b_i, i \in \{1, \ldots, \beta\}$ two values are maintained: the count c_i that corresponds to the number of values that fall in bin b_i, and the mean m_i that is defined to be the mean of all values belonging to the same bin b_i. Figure 5 outlines the code of maintaining the statistics of the bins of a histogram build over the requested values of one attribute, i.e., its predicate set. The min value of the domain, the width w, and number of bins β are considered to be known beforehand. The variable N contains the total number of values that have been observed in the predicate set.

The statistics of the histograms provide the means to determine the distribution of the interest, and based on them, a weight is assigned to each newly appended tuple. We adjust the kde function to consider only the mean values m_i of the β bins multiplied by the count c_i instead of iterating over all observed values $x_1, \ldots, x_N$. The resulting estimator function is now defined as:

$$\breve{f}(x) = \frac{1}{N \times w} \sum_{i=1}^{\beta} c_i \times \phi\Big(\frac{x - m_i}{w}\Big)$$

where β is the total number of bins, w the width of the bins, and c_i the count and m_i the mean of the i-th bin. Since $\beta \ll N$, and β is fixed, $\breve{f}(x)$ can be computed in constant time. Also

$$\int K_w(u) = 1 \text{ and } \textstyle\sum_{i=1}^{\beta} c_i = N \Rightarrow$$
$$\textstyle\int \sum_{i=1}^{\beta} c_i K_w(u) = N \times \int K_w(u) = N \Rightarrow$$
$$\textstyle\int \breve{f}(x) = N^{-1} \int \sum_{i=1}^{\beta} c_i K_w(u) = N^{-1} \times N = 1.$$

Thus, function $\breve{f}$ is an estimation of the probability density function that describes the relative likelihood for value x to occur in the predicate set. The purple line of Figure 4 shows the density function computed with $\breve{f}$. It is almost identical with the estimation from $\hat{f}$, while it only iterates over a few constant number of bins, and the bandwidth is always equal to the width of the bins.

Assume a newly ingested tuple t_{new} during incremental load. For simplicity, assume also that t_{new} has only one attribute of interest [4]. A weight is assigned to tuple t_{new} equal to $\breve{f}(t_{new})$. We bias the sample by making the probability of choosing this tuple for an impression proportional to $\breve{f}(t_{new}) \times N$. Function $\breve{f}$ estimates the frequency of appearance of value x in the predicate set. Thus, the more frequent the value, the larger the product $\breve{f}(t_{new}) \times N$, and the higher the probability of choosing t_{new}.

Function $\breve{f}(x)$ can be used in the reservoir setting. An impression has always a predefined size n, thus for a uniform sampling a tuple is accepted with probability n/cnt, where cnt is the number of tuples in the database. For biased sampling the probability of accepting a tuple t is $\breve{f}(t) \times N$, and by normalising this with the desired size of the impression leads to the following probability

$$P(\text{accept } t) = \breve{f}(t) \times N \times \frac{n}{cnt}$$

where N is the size of the observed predicate set, n the size of the desired impression, and cnt the number of tuples in the database. Figure 6 details the biased sampling reservoir algorithm. After a tuple is accepted, another randomly chosen one is thrown out from the sample to make room for the new.

The leftmost histograms of Figure 7 show the distributions of the values of the base data of SkyServer answering the queries used in Figure 4 (more than 600.000 tuples). We create two impressions of 10.000 tuples for each attribute: one based on uniform sampling (red histograms of Figure 7), and one based on biased sampling (purple histograms of Figure 7) steered by the interest shown in Figure 4. The impression created with bias contains many more tuples from the areas of interest, achieving a better representation of data around the focal points.

```
populate the sample smp with the first n tuples;
cnt := n;
while (tpl := block_until_next_tuple())
      cnt++;
      rnd := random();
      if ((cnt*rnd) < (n*N*f̆(tpl)))
         smp[floor(rnd*n] := tpl;
      end
end
```

Figure 6: Biased Sampling reservoir algorithm

5. RELATED WORK

Various techniques on how to construct data synopses, keep summary statistics, and obtain data samples have been proposed in the past [5, 8, 10, 19, 23]. A topic of intense research is how samples can be adjusted to support correlations between join attributes [3, 4, 18, 21]. SciBORQ is also aiming towards efficient inter-column and inter-table sampling. Self-tuning samples were proposed by ICICLES [7]. The results of a query are regarded as newly ingested

proaches are part of future research.

[4] multiple attributes in the same tuple are dealt either with multi dimensional histograms or with a combine function $c(t_{new}) = \breve{f}(t_{new}.att_1) \circ \cdots \circ \breve{f}(t_{new}.att_m)$.

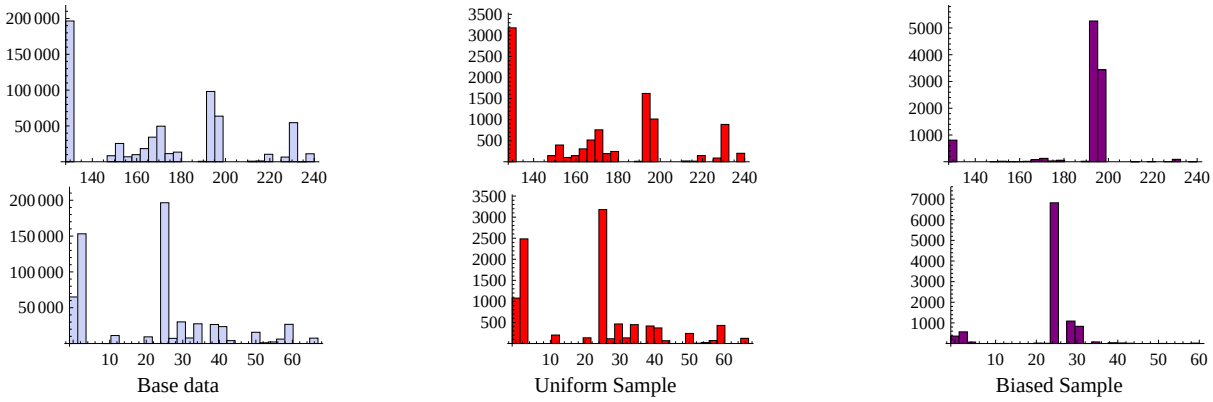


Figure 7: 1st row is for predicate `'ra'` and 2nd row for `'dec'`

data, and the sample is updated accordingly. We intend to investigate this technique for SciBORQ also: a side-effect of a query evaluation is to update an impression using query results. Another tuning approach for histograms was proposed in [1], where the feedback of a query is used to refine histograms to better resemble the base data. Gibbons and Matias envisioned a system similar to ours in their motivation for concise samples [9]. This led to Aqua [2], a system for providing approximate answers to aggregate queries. Both of those are close to our vision, however, they lack the multi-layer design and adaptive biased sampling of SciBORQ that allows the system to adjust the quality guarantees during query execution.

6. SUMMARY AND FUTURE WORK

In this paper we described a new data exploration architecture for science data warehouses. The key observation is that in most situations a fraction of the data would be a good starting point, provided that the error and processing time bounds are within the requested range.

Biased sampling is a valuable alternative to the predominant uniform sampling techniques, since more data from the areas of interest are sampled. The architecture of SciBORQ is unique in its multi-layer approach, providing the means for runtime execution and (re-)optimisation that will guarantee the desired error bounds, even if they are thrown off track due to correlations. We intend to investigate the theoretical error margins for biased sampling based on known mathematical tools [6] and their propagation through the fundamental query processing operators, and to incorporate multi-dimensional histograms for sampling over relations. Finally, we will explore the connection between query processing time, the size of an impression, and the consumption of resources.

7. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *Proc. of the ACM SIGMOD*, 1999.

[2] S. Acharya, P. Gibbons, and V. Poosala. Aqua: A Fast Decision Support System Using Approximate Query Answers. In *Proc. of the 25th VLDB*, 1999.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proc. of the ACM SIGMOD*, 1999.

[4] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. *ACM SIGMOD Record*, 28(2), 1999.

[5] Daniel Barbara et al. The New Jersey Data Reduction Report. *IEEE Data Eng. Bull*, 20(4), 1997.

[6] A. Fog. Sampling Methods for Wallenius' and Fisher's Noncentral Hypergeometric Distributions. *Communications in statistics, Simulation and Computation*, 37(2), 2008.

[7] V. Ganti, M. L. Lee, and R. Ramakrishnan. ICICLES: Self-Tuning Samples for Approximate Query Answering. In *Proc. of the 26th VLDB*, 2000.

[8] M. Garofalakis and P. B. Gibbons. Probabilistic wavelet synopses. *ACM-TODS*, 29(1), 2004.

[9] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. of the ACM SIGMOD*, 1998.

[10] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM-TODS*, 27(3), 2002.

[11] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[12] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. In *Proc. of the 3rd CIDR*, 2007.

[13] M. Ivanova, M. Kersten, N. Nes, and R. Goncalves. An Architecture for Recycling Intermediates in a Column-store. In *Proc. of the ACM SIGMOD*, 2009.

[14] M. C. Jones, J. S. Marron, and S. J. Sheather. A Brief Survey of Bandwidth Selection for Density Estimation. *Journal of the American Statistical Association*, 91(433), 1996.

[15] R. A. Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of XQueries. In *Proc. of the ACM SIGMOD*, 2009.

[16] S. Manegold, M. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. In *Proc. of the 35th VLDB*, 2009.

[17] MonetDB. http://monetdb.cwi.nl.

[18] M. Muralikrishna and D. J. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *Proc. of the ACM SIGMOD*, 1988.

[19] F. Olken and D. Rotem. Simple Random Sampling from Relational Databases. In *Proc. of the 12th VLDB*, 1986.

[20] E. Parzen. On Estimation of a Probability Density Function and Mode. *Annals of Mathematical Statistics*, 33(3), 1962.

[21] V. Poosala and Y. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proc. of the 23rd VLDB*, 1997.

[22] A. Szalay and R. Brunner. Exploring Terabyte Archives in Astronomy. Invited talk at the IAU Symposium in Baltimore, 1996.

[23] Viswanath Poosala and Venkatesh Ganti and Yannis E. Ioannidis. Approximate Query Answering using Histograms. *IEEE Data Eng. Bull*, 22(4), 1999.

[24] J. S. Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1), 1985.