A Production-Quality Driver for Berkeley DB Using the Erlang Driver Toolkit

chris.newcombe@gmail.com

Overview

- Motivation
- Overview of EDTK
- Status of EDTK v1.1 (the last release)
- Work done
  - EDTK bug fixes
  - EDTK enhancements
  - BDB driver bug-fixes
  - BDB driver enhancements

Why Berkeley DB?

- In-process library (statically or dynamically linked)
- Excellent code quality – very low risk of destabilizing Erlang VM
- Extremely flexible
  - ACID transactions (inc. fast recovery)
  - High performance and highly concurrent (even for writes)
  - 256 TB per-node data size limit
  - Single-master replication/failover (to any number of replicas)
  - Distributed transactions if you need them
- Sleepycat Support is literally the best in the industry
- Proven in many commercial mission-critical systems
- Previous good experience with BDB-HA (replication)

Why not Mnesia/DETS?

- DETS recovery time
- DETS 2GB per-node limit
- Write-concurrency
- Mnesia Fragmented Tables rely on Distributed Erlang, but we might need to scale beyond Distributed Erlang (to 1000's of nodes)
- BDB has a known operational profile (performance tuning, failure modes).

Overview of Erlang Driver Toolkit
(written by Scott Lystig Fritchie)

- Declarative API wrapper / code-generator
- Generates marshalling code for commands and responses
  - uses outputv and driver_output_term, so linked-in drivers can avoid copying binaries
- Tracks resources allocated by the wrapped library (“valmaps”), and does automatic clean-up on exit
- Hides the difference between linked-in drivers and pipe-mode drivers
  - this adds important operational flexibility
- Saved us a HUGE amount of time. Thanks Scott!

Status of EDTK v1.1 examples/berkeley_db

- BDB chosen due to it's large API, to drive EDTK features
- Written for Berkeley DB v4.0 : now a very old version
  - constants in berkeley_db.xml need to change for each new BDB release
- Many BDB APIs wrapped but some important ones missing
  - No transaction support due to potential deadlock issues
  - No house-keeping/admin APIs (checkpointing, statitistics)
  - No replication support (was alpha-quality in BDB v4.0)
  - Incomplete support for very useful APIs like db_get(CONSUME_WAIT)
- Some bugs found in our testing
- Some safety issues : e.g. 'valmap' tuples can be accidentally re-used (causing seg-faults)

Path to Production Quality

- **EDTK's presumed concurrency model is not a good match for BDB**
- We certainly need transactions, so address deadlock issues
- We need nested transactions, which requires valmap support (parent/child relationships)
- Need db_get(CONSUME_WAIT) – transactional message queues are very useful
- Need flexibility at runtime (e.g. avoid re-compile/ re-deploy to change an arbitrary limit)

Quick Overview of BDB Objects (managed by EDTK valmaps)

- **DB** : A “Database” (analogous to a table in relational systems)
  - Stores arbitrary binary (untyped) key->records
  - Typically corresponds to a single physical file (page-oriented)
  - Typically a BTREE (others are QUEUE, HASH, RECNO)
- **DB_ENV** : A “Database “Environment” (analogous to a relational database)
  - A set of databases, transaction logs, and ‘region’ files
  - ‘region’ files are IPC structures, plus a page-cache (critical for performance)
  - Largest scope of normal transactions (i.e. transactions can span databases)
  - Unit of replication (as it is done by log-shipping)
- **DB_TXN** : a multi-step transactions (can also do auto-commit operations)
  - single-threaded (the application *must* enforce this; EDTK v1.1 does)
  - multi-step transactions hold locks between the steps
- **DBC** : a cursor handle into a single database
  - single-threaded (the application *must* enforce this ; EDTK v1.1 does)
  - holds a lock on the page it is on
  - may be opened within a multi-step transaction (which will accumulate locks)

Concurrency Issue #1: Erlang processes and Port instances

- Need to use BDB from a large number of Erlang processes
- EDTK v1.1 assumes that each Erlang process will open its own instance of the port
- May be fine for some libraries, but not good for BDB:
  - BDB startup and shutdown must be coordinated (single threaded), not a free-for-all.
  - Likely need to limit number of Erlang processes using BDB at any one time (avoid thrashing, huge queues/backlog
  - Port instances cannot share valmaps : Each Erlang process would open it's own DB_ENV and DB handles
  - Some BDB objects (DB_ENV, DB) are expensive to create, and are designed to be shared (thread-safe)
  - DB_ENV and DB handles open files. Many Erlang processes * many databases could easily exhaust file-descriptors
  - Some BDB features don't allow multiple DB_ENV handles per OS process (DB_REGISTER, replication)
  - In pipe-mode, every port instance spawns a new OS process

But Automated Clean-Up is Vital

- An Erlang process may crash while holding open cursors and/or transactions, which may hold locks
- If not closed/aborted, the locks will deadlock other BDB threads
- The 1-to-1 Erlang/Port model guarantee clean-up if the Erlang process dies (or if the port dies)
- Any alternative must do likewise

How about a server?

- Fewer open ports (ideally 1) implies a server
- Pros:
  - Can coordinate BDB startup/shutdown
  - Can link to clients to know to do clean-up
  - Can cache expensive DB handles (**but valmaps will need to support concurrent access)**
  - Can handle BDB configuration housekeeping, replication etc.
  - Could even abstract BDB entirely
    - BDB's API is sometimes dangerous: e.g. a single txn_commit/abort with an open cursors will panic the entire environment

How about a server?

- Cons:
  - Server must either spawn an Erlang worker per client and hand-off the session, or issue *non-blocking* BDB commands on behalf of clients and forward the replies to clients (and handle timeouts etc)
    - Latter case implies lots of state to track/update (for hundreds/thousands of clients)
  - To do per-client cleanup, server needs to track which transactions/cursors are owned by that client
    - Somewhat duplicates EDTK's valmap code.
  - A second complex API to write (or generate with EDTK) and maintain.
  - Performance: All commands and responses go through an extra hop. (Probably not an issue, but nice to avoid if it's easy.)

How about half-a-server? The PortCoordinator

- EDTK v1.1 uses driver_output_term to reply to Erlang.
- Browsing the documentation, I noticed
  - driver_send_term : sends a message to an arbitrary Erlang pid
  - driver_caller : returns pid of process that send the current command to the driver
- These remove the need for BDB commands/replies to go through the server
  - The port instance itself is now almost a BDB server
  - No need to write/maintain another API
  - No extra hop for commands/replies
- But we still need a bit of a server on the Erlang side:
  - Startup/shutdown coordination, house-keeping etc.
  - API : get_port(), release_port()
  - API : get_db_handle()
  - Clean-up somehow if clients crash

PortCoordinator Problem #1

- If clients send txn_begin etc. directly to the port, how does the server do clean-up if a client crashes?
- Solution : leverage EDTK's code
  - get_port() links to client process and returns a unique 'group valmap id' (an unsigned int, wraps at 2^32 and that's OK).
  - All EDTK valmap "start" operations require a group_valmap_id parameter
  - The C code stores the group_valmap_id with the valmap value
  - A new EDTK command 'group_valmap_cleanup' cleans-up all valmap entries (of any valmap type) owned by the same group-id. Implementation is a bit tricky as :
    - some of the operations to be cleaned up may still be in-progress or enqueued for later processing
    - valmaps must be cleaned-up in order of type (all DB_TXN before DB)
    - we don't want to block the Erlang VM thread
- Annoyance: clients MUST pass the right group_valmap_id, and must NOT share it with other processes. Solved later.

PortCoordinator Problem #2

- In pipe-mode, driver_caller and driver_send_term are are not available
  - Write a stub for driver_send_term that just calls driver_output_term
  - All replies go to the port's connected process (i.e. the server).
  - The server needs to forward the replies to clients, but how does it know where to send them?
- Solution: tag all commands
  - Tag is currently {Ref, SenderPid}
  - The ref. allows safe non-blocking use of BDB commands, as we can always associate replies with the correct command.
  - EDTK does not support marshalling of refs or pids.
    Fortunately the tag should be treated as an opaque token, so I just run the tag through term_to_binary.
    So in linked-mode the tag is not even copied

Concurrency Issue #2
Multiplexing Erlang Processes to OS Threads

- BDB is obviously IO-intensive
- BDB uses blocking IO (for portability)
- Core BDB does not create threads of it's own, it blocks the caller's thread
- But not a good fit for the Erlang VM's async threadpool
  - Some BDB operations block for a very long time, or unbounded time -- e.g. db_get(CONSUME_WAIT).
    This would play havoc with essential Erlang drivers like erl_file that use the VM's async threadpool.
  - Also need to avoid several kinds of self-deadlock
- In pipe-mode, it obviously blocks the whole external OS process (no thread-pool)

Other threading issues: self-deadlocks due to BDB locks

- BDB multi-step transactions were designed for thread-per-transaction
- Architectures based on worker thread-pools can use them, but must be careful if the threads have queues:
  - Erlang process P1 owns a transaction handle that owns a BDB lock on a database page
  - Erlang process P2 attempts to do a BDB read/write that blocks on that BDB lock. The operation blocks worker-thread T.
  - Erlang process P1 attempts to commit or abort it's transaction (release the BDB-lock). If this operation happens to be enqueued so that only worker-thread T can process it, then we have a self-deadlock
  - BDB has a built-in deadlock detector for resolving deadlocks amongst it's *own* locks. That mechanism does *not* help in this case.

Overview of Erlang VM Async Thread-Pool

- Single pool shared by all drivers that call driver_async
- Static number of threads (‘erl +A50’)
- Each thread has it’s own producer/consumer queue
- By default, work is assigned to threads on a round-robin basis.
  - Implies some threads may be idle while others have items waiting in their queues (inefficient)
- driver_async has a ‘int * key’ parameter. If non-null then it chooses a thread based on *key
  - e.g. erl_driver uses (or may one-day use) the file-descriptor as *key, so all operations on the same file are serialized.
  - mapping algorithm is not documented: in R10B it is a simple *key modulo number-of-threads.
- Obviously not available to pipe-drivers

Multiplexing option: #1 driver_async with appropriate *key

- If every transaction had it's own thread, the previously-described deadlock could not happen
- EDTK stores transaction handles in a 'valmap' array – a plain C array. So we could use the array index as *key.
- But it is still has major problems
  - Cannot size thread-pool at runtime, so **requires** that 'erl +A' specifies at least as many threads as concurrent transactions (specified in berkeley_db.xml), or could still deadlock.
  - Other BDB operations do IO, and may block for any amount of time: txn_checkpoint, memp_trickle, db_get(CONSUME_WAIT)
  - When replication is enabled, threads executing txn_commit/abort block until replicas acknowledge receipt of the update
  - Critical Erlang drivers share the same thread-pool, and so may block for a long time

Multiplexing #2: Private Driver Threadpools

- Those problems would disappear if each port instance had its own private threadpool
- Perfectly legal
  - Don’t call any Erlang VM code from worker threads
  - Use driver_select to register a file descriptor. Write to the file descriptor when we have responses ready.
  - When the fd becomes writable, the VM calls our ready_input callback. A small bit of code calls our ready_async callback (just as the VM would do), to send the reply back to Erlang.
  - Looking at the VM’s erl_async.c code, this is basically what the VM’s implementation does ☺
- While we’re at it, solve some of the problems
  - One queue feeds all of the threads. Threads consume the next item as soon as they finish the last. No more idle threads if there is work available, and no deadlocks due to fixed-choice of threads.
  - Add driver command to resize the thread-pool at runtime (ops. flexibility)

Private thread-pools in pipe-driver mode

- Restores feature parity with linked-in mode.
- Before:
  - EDTK v1.1: N Erlang processes using BDB would have spawned N external OS processes.
  - EDTK with PortCoordinator server: 1 server (with N clients) has 1 port instance, which has 1 external OS process
    - Hopeless performance and almost immediate self-deadlock
- Now that pipe-main supports private thread-pools, the PortCoordinator threading model is fine (the same)
- Not much code. Main thread pretends to be VM:
  - implements a driver_select stub, and stores the fd
  - selects on stdin and the fd registered by driver_select
  - calls outputv for data from stdin, and ready_input if fd is readable
  - reference-count driver binaries and free them as necessary

But still some deadlocks...

- Slightly different scenario/reason
  1. The thread-pool has N threads
  2. Erlang process P0 owns a transaction handle that owns a BDB lock on a database page
  3. Erlang processes P1..PN do read/write operations that happen to block on that BDB lock. i.e. Those operations tie-up (block) all worker threads.
  4. No further operations are possible, including committing/aborting the transaction that owns the lock.
  5. Increasing #threads is obviously not guaranteed to solve it (other operations may fill them)

Solution Part #1
Multiple private thread-pools per port instance

- Individual driver commands can choose a thread-pool via an XML attribute. (Default is thread-pool #0)
- All thread-pools can be resized dynamically and independently.
- BDB driver currently uses 4:
  - #0 : General workers (e.g. put/get/misc. operations).
  - #1 : All valmap 'stop' operations; e.g. txn_begin, txn_abort, c_close, group_valmap_cleanup. Also, explicit deadlock detection if configured. i.e. Anything that can release locks without obtaining locks itself.
  - #2 : db_get(CONSUME_WAIT). This can block for arbitrary time, so we don't want to tie up general worker threads.
  - #3 : House-keeping operations that might take a long time; e.g. txn_checkpoint, memp_trickle (the latter flushes dirty pages to disk to minimize IO-intrusiveness of checkpoints).

There is still a problem: Solution Part #2

- Multiple thread-pools allows P0 to commit or abort. But if it does another put/get, then it still deadlocks (because it uses thread-pool #0, which is totally blocked)
- Option #1: Have a dedicated thread for each transaction as before (i.e. N thread-pools each with 1 thread, selected by valmap db_txn index).
  - We have solved the original problems with that approach:
    - As we own the thread-pools, we don't need to gamble on 'erl +A'
    - We own the selection function (modulo/hashing)
    - Can still have dedicated pools for db_get(CONSUME_WAIT), house-keeping. etc.
  - But this is hugely wasteful solution for a very rare condition (all threads blocked on locks). If we want to allow 1,000+ concurrent BDB transactions at peak, we need 1,000+ OS threads.

Solution Part #2 : timeouts

- Option #2: Size thread-pools based on load/performance testing, and use per-operation timeouts to detect rare problems
  - Timeout kills the blocked Erlang process, so the port coordinator runs group_valmap_cleanup, which releases locks.
  - Fixes the deadlocks, but may thrash against the limit when overloaded.
    May want to detect that condition and increase #threads, reduce max #transactions, reduce max #port coordinator clients, etc.
- Also implemented a limit on each thread-pool's queue length, to detect totally-stuck drivers before they fill memory.
  - If BDB get's stuck due to a bug, reboot the Erlang node
  - Again, limit can be changed at runtime

Specifying timeouts: default arguments

- We don’t want to add a timeout argument to all EDTK calls; there are too many arguments already.
- A reasonable default will do, but we do need to be able to override it;
  e.g. should be larger for txn_checkpoint, infinity for db_get(CONSUME_WAIT),
- Solution: Turn the Port argument into a PortHandle record
  - Application code never uses it as a plain Erlang port anyway (EDTK stubs do that)
  - We can give it a useful type (based on the driver name), so callers can use is_record(P, bdb_port_handle) in guards
  - We can also store the group_valmap_id in there
    - Hides it from clients (who shouldn’t mess with it or share it)
    - Avoids changing the signature of valmap “start” calls when the group_valmap_cleanup feature is enabled.
  - We can allow the driver to put other default arguments in there
    - BDB driver uses this for DB_ENV (as only 1 is allows per port)

Sections to add #1

- Generation-ids in valmap tuples
- Port-ids in valmap tuples to prevent crashes
- Implementation of concurrent (ref-counted) valmaps (DB_ENV and DB)
- Parent/Child valmap relationships (nested transactions)
- is_record for valmaps
- is_record for PortHandle
- bdb_stop() is fixed and made parallel to minimize blocking of the Erlang VM (as all valmaps must be freed before it returns and txn_abort may do IO/replication).
- port_coordinator config handling, and house-keeping
- port_coordinator:terminate/2 still does most cleanup manually (using group_valmap_cleanup_non_block), again to reduce blocking.
- db_get(CONSUME_WAIT)
- berkeley_msg_queue
- berkeley_sequence_server
- Replication (and event_notify)

Sections to add #2: Fixed EDTK v1.1 bugs

- stop didn't acquire the mutex when accessing valmap arrays, but operations may still be in-flight
- stop doesn't wait for in-progress operations involving first valmap type to finish before it starts closing valmaps of the second type (and so on)
- value returned by wrapped library is leaked if no valmap slots are free after the call (solution was to reserve the slot when the command is received)
- BDB driver installed allocation fundtions that called driver_alloc_binary in async workers (from within BDB code). This resulting in memory corruption and segfaults under concurrent load (esp when other Erlang processes were doing stuff like ETS inserts).
- stop() did not call cleanup_.._index() for any valmap entries that are INUSE, so the code in cleanup_.._index() that sets DELAYED_CLEANUP is never activated, so some valmaps (that are in-use when stop is called) are never cleaned up. (DELAYED_CLEANUP would self-deadlock if it was ever used, as cleanup_index() was called while desc->mutex was held, and cleanup_index tried to acquire the (non-recursive) mutex again).
- stop() needs to enable auto-BDB-deadlock detection, as if the app was calling the explicit deadlock detector frequently, it can nolonger do so once stop has been called, but operations might still be in progress that could deadlock
- successful valmap 'start' operation is performed after stop is called (the async_free call is currently just sys_free (to free the callstate object), so nothing currently calls the valmap cleanup_func on the result of the library call made by invoke -- i.e. an object returned from the library (bdb) is leaked)
- stop() chewed up stack if it was not safe to return (valmaps still in use)
- cleanup_index held mutex while calling cleanup function – which may take a long time.
- Romain Lenglet's patch

More Fixed EDTK v1.1 bugs

- dbenv_close and dbenv_remove now flagged as global-exclusive operations. These are not allowed if any valmaps are in-use at all. (Not good enough – needs to be, if any operations are happening; not quite the same thing). Also, BDB drive code refuses to do these if any valmap db, txn, cursor entries are allocated.
- db_rename, db_remove, db_upgrade were not tagged as valmap "stop" operations, so the Db handle was left in the valmap array, and would crash during _stop().
- removed dbenv_set_errpfx as it made BDB refere to memory that was freed
- read_ansync would free binaries owned by callstate. But, after _stop() is called, _ready_async won't be called on any completed operations – the 'async_free' function passed to driver_async will be called instead. This was set to sys_free, so the binaries would be leaked. I made a new free_callstate function.