Berkeley DB: Architecture

WRITTEN BY KARAN SIKKA

In my quest to learn more about how databases work, I started with Berkeley DB for one reason: simplicity. It doesn't parse SQL, it doesn't create a query plan, it doesn't have a client/server architecture. It's just a key-value store that you `#include` in your code. Intuitively it seems like it could be a building block in constructing a full blown relational database server.

What is it?

BDB is a persistent, key-value store that's unaware of the underlying type you're storing. It stores your data in a single file which it manages automatically. It supports persistent `get`, `set`, `insert`, `delete` on top of one of four highly-performant data structures. It even supports transactions and replication.

Rather than connecting to BDB like you would to MySQL, BDB is a C library that links directly to your application. The first advantage of

doing this is lower administrative overhead. You don't need to stand up a MySQL database, make sure it's reachable, deal with users and permissions, etc. The second is lower IO overhead. You don't need to talk to BDB through a socket. All you need is a function call.

How does it work?

BDB stores, organizes, and retrieves data from a file on disk. There's a single file for each database, which BDB slices up into multiple "pages" for its own internal use. Each page has a page id. Pages can be created, deleted, read to cache, released from cache, and flushed back to disk.

BDB data structures are implemented in such a way that they read and write on these pages in memory. The task of commanding these pages to write back to disk is that of the Transaction manager.

When you modify a B+ tree or Hash structure, your writes are typically not sequential. You may jump from pointer to pointer, writing on non-adjacent pages. This is slower than writes to sequential pages on hard disk drives. Thus, as a performance optimization, BDB will avoid flushing pages to disk immediately, and will instead write out what changed in a sequential manner to a log database.

There's a risk that the process goes down before the database state is consistent with the log state. BDB can use the log to recover

the database.

BDB supports highly concurrent access from multiple threads and multiple processes. It uses page-level locking to ensure data isn't corrupted when being accessed concurrently. Modern databases may implement record-level locking for more granular concurrency, but BDB does not for simplicity's sake. And in practice it works quite well. If you need more granular locking, you can reduce the page size.

Phew, that's a lot of responsibility for BDB to handle. BDB breaks it down into four subsystems:

1. **Buffer manager:** A page-based cache for persistent data structures on disk.
2. **Lock manager:** Safely allows concurrent data access accross threads and processes.
3. **Log manager:** Code to sequentially store "breadcrumbs" for DB operations.
4. **Transaction manager:** Uses Locks and Log to implement transactions and recovery.

Now I'll summarize what each subsystem does and how it works. Or, you can skip the deep dive and move on.

Deep(er) dive

1. The buffer manager: Mpool

BDB data structures are not directly implemented on memory, nor are they directly implemented on files. The data structure "pointers" are not memory addresses; rather, they're page numbers (and maybe an offset into the page?). A "page" is a discrete chunk of memory that you can request from Mpool. There's probably some limit to the number of pages you can be operating on at one time, but there's virtually no limit on the number of pages you can have total, allowing the total database size to be greater than the amount of available memory. Pages live on disk, and when you request a page, Mpool reads it from disk and caches the page. Future writes occur in memory only until you call "flush" on the page. Pages can be pinned to memory to prevent eviction, and unpinned to allow cache eviction.

Mutating data structures in memory means that there is risk of data loss if the process crashes before the page is flushed. However, if you turn on Write Ahead Logging, BDB will write and persist a log entry before the operation occurs. This feature does hurt performance, since the log page will need to be flushed on every operation. The Log and Transaction sections explain this in more detail.

2. The lock manager: Lock

BDB uses page-level locking to ensure atomic operations. I infer this to mean that when you're mutating a data structure on a page, you acquire

a lock so that others must wait till you're done writing to read the page. Similarly, if you're reading the page, you want to enforce that people can't write to the page until you're done reading. However, if you two operations want to read from a page, they may do so concurrently. The rules for which locks conflict are encoded in a "Conflict Matrix".

The Conflict Matrix is an efficient way to separate concerns. Locking code doesn't need to be concerned with the specific lock configuration. And describing the lock configuration shouldn't be intertwined with core locking code. Separating the two makes it easier to change the configuration without worrying about introducing nasty bugs. It also makes it easier to test the core Lock logic.

3. The log manager: Log

The log is a data-structure that sequentially stores the transational operations that occured, for the purpose of aiding Recovery. If the database crashes while changes to in-memory buffers have not yet been flushed, a persisted record of the changes that built up to that point provides the information necessary to recover. Without the log, the BDB loses the ability to implement the D in ACID (durability). More about recovery in the transaction manager section.

The log is implemented on top of Mpool just like a database Btree would be, which justifies the decision to make Mpool is a separate module.

4. The transaction manager: Txn

Transactions are what allow BDB support ACID. Full ACID transaction support is optional in BDB - you can turn it off for faster performance, or on for greater ACIDity. When you turn transactions on, every DB query is automatically wrapped in a transaction, and as mentioned before, BDB writes a durable log entry before a transaction begins. Writing to the log in a persistent fashion is one of the reasons for the ACID/performance tradeoff.

Txn is responsible for taking checkpoints - though it's not clear from the article I read on what schedule. Every so often? After some number of txns? Taking a checkpoint means forcing all dirty Mpool buffers to flush and writing a checkpoint log record that says "the pages on disk accurately reflect all the txns up till now".

Recovery is a conceptually simple operation. From the last txn log record, go backwards in the log "undoing" any txns which were BEGIN but not COMMIT. This rectifies the state for all uncommitted or aborted transactions. Then go forward from the checkpoint to the end of the log, "doing" any txns which were committed. Now the state is correct. Take a checkpoint to prevent this work from happening again.

It bothers me that I don't fully understand the implementation of "undoing" and "doing" above. I'm not sure how BDB would handle random data corruptions.

Learning more

Here's my process for how I came to know this information.

First I read the Berkeley DB article from The Architecture of Open Source Applications, which provided insight into the reasoning behind architectural decisions. There were lots of great architecture tips along the way. I started writing the first draft of this, during which I had to re-read the article to get a deeper understanding.

To fully answer my questions, I had to read the documentation, which was *very* well written. It laid so much information about BDB in such an efficient manner, I was in awe. I'm sure I'll revisit it again sometime.

Alternating between the docs and the architecture paper would to be the best way to learn more about BDB.

Final thoughts

For a first pass, this was fruitful. I learned a lot about the general decisions that BDB made, and it generated a lot of questions.

Out of curiosity, I googled "BDB Write ahead log", and stumbled upon a SQLite page! It turns out SQLite uses a different method of recovery, but

supports WAL optionally. I'll read about that sometime, but next up on my queue is A Comparison of Approaches to Large Scale Data Analysis.

Huzzah! We're at the end. I think I will do this again, but I'm not sure if writing write-ups this long is sustainable. I hope it gets easier with practice. I'll play it by ear. Until next time!

« Full blog

Let's get in touch! karanssikka@gmail.com