LevelDB - Fast and Lightweight Key/Value Database From the Authors of MapReduce and BigTable

Wednesday, August 10, 2011 at 9:01AM

Todd Hoff in Product, databases

LevelDB is an exciting new entrant into the pantheon of embedded databases, notable both for its pedigree, being authored by the makers of the now mythical Google MapReduce and BigTable products, and for its emphasis on efficient disk based random access using log-structured-merge (LSM) trees.

The plan is to keep LevelDB fairly low-level. The intention is that it will be a useful building block for higher-level storage systems. Basho is already investigating using LevelDB as one if its storage engines.

In the past many systems were built around embedded databases, though most developers now use database servers connected to via RPCs. An embedded database is a database distributed as a library and linked directly into your application. The application is responsible for providing a service level API, sharding, backups, initiating consistency checking, initiation rollback, startup, shutdown, queries, etc. Applications become the container for the database and the manager of the database.

Architectures using embedded databases typically never expose a raw database abstraction at all. They have a service API and the services use the embedded database library call transparently behind the scene. Often

an embedded database will provide multiple access types, like indexed access for key-value uses and btrees for range queries and cursors.

BerkelyDB is one well known example of an embedded database, SQLite is another, the file system is perhaps the most commonly used database, and there have been many many other btree libraries in common use. I've used C-tree on several projects. In a battle of old versus new, a user named IM46 compared Leveldb to BerkelyDB and found that LevelDB solidly outperforms Berkeley DB for larger databases.

Programmers usually thought doing this stuff was easy, wrote their own failed on-disk btree library (raises hand), and then look around for a proven product. It's only relatively recently the databases have gone up market and included a network layer and higher level services.

Building a hybrid application/database architecture is still a very viable option when you want everything to be just so. If you are going to load balance requests across sharded application servers anyway, using a heavy weight external database infrastructure may not be necessary.

The LevelDB mailing list started off very active and has died down a bit, but is still nicely active and informative. Here are some excellent FAQish tips, performance suggestions, and porting issues extracted from the list:

- Largest tested database: 1 billion entries with 16 byte keys and 100 byte values (roughly 100 GB of raw data, about half that after compression).
- LevelDB has been Open Sourced.
- Relationship between LevelDB and BigTable: The implementation of LevelDB is similar in spirit to the representation of a single Bigtable tablet (section 5.3). However the organization of the files that make up the representation is somewhat different and

is explained [in source code comments]. They wanted to put together something like the BigTable tablet stack that had minimal dependencies and would be suitable for open sourcing, and also would be suitable for use in Chrome (for the IndexedDB implementation). LevelDB has the same general design as the BigTable tablet stack, but does not share any of the code.

- Didier Spezia on log-structured-merge (LSM) trees: They are mostly useful to optimize random I/Os at insertion/delete time at the price of a slightly degradation of read access time. They are extremely efficient at indexing data in random order stored on rotational disks (i.e. better than b-trees).
- Optimized for random writes. TokyoCabinet could be filled with a million 100-byte writes in less than two seconds if writing sequentially, but the time ballooned to ~2000 seconds when writing randomly. The corresponding slowdown for leveldb is from ~1.5 seconds (sequential) to ~2.5 seconds.
- In the tradition of BerkelyDB it's a library you embed in your program, it's not a server. You will have to add the networker layer, sharding etc if a single process won't suffice.
- Quite appropriately threading decisions are left to the application, the library is not thread safe. Threads sharing iterators, for example, will need to lock.
- Data is written in sorted order.
- C++ only.
- Variable sized keys are used to save memory.
- What leveldb does differently from B+trees is that it trades off write latency for write throughput: write latency is reduced by doing bulk writes, but the same data may be rewritten multiple times (at high throughput) in the background due to compactions.
- Log-Structured Merge Trees offer better random write performance (compared to btrees). It always appends to a log file, or merges existing files together to produce new ones. So an OS crash will

cause a partially written log record (or a few partially written log records). Leveldb recovery code uses checksums to detect this and will skip the incomplete records.

- Search performance is still O(lg N) with a very large branching factor (so the constant factor is small and number of seeks should be <= 10 even for gigantic databases).
- One early user found performance degraded at around 200 million keys.
- Bigger block sizes are better, increasing the block size to 256k (from 64k).
- Batching writes increases performance substantially.
- Every write will cause a log file to grow, regardless of whether or not you are writing to a key which already exists in the database, and regardless of whether or not you are overwriting a key with the exact same value. Only background compactions will get rid of overwritten data. So you should expect high cpu usage while you are inserting data, and also for a while afterwards as background compactions rearrange things.
- LevelDB Benchmarks look good:
  - Using 16 byte keys at 100 byte values:
    - Sequential Reads: LevelDB 4,030,000 ops/sec; Kyoto TreeDB 1,010,000 ops/sec; SQLite3 186,000 ops/sec
    - Random Reads: LevelDB 129,000 ops/sec; Kyoto TreeDB 151,000 ops/sec; SQLite3 146,000 ops/sec
    - Sequential Writes: LevelDB 779,000 ops/sec; Kyoto TreeDB 342,000 ops/sec; SQLite3 26,900 ops/sec
    - Random Writes: LevelDB 164,000 ops/sec; Kyoto TreeDB 88,500 ops/sec; SQLite3 420 ops/sec
  - Writing large values of 100,000 bytes each: LevelDB is even Kyoto TreeDB. SQLite3 is nearly 3 times as fast. LevelDB writes keys and values at least twice.
  - A single batch of N writes may be significantly faster than N

individual writes.
    - LevelDB's performance improves greatly with more memory, a larger write buffer reduces the need to merge sorted files (since it creates a smaller number of larger sorted files).
    - Random read performance is much better in Kyoto TreeDB because it cached in RAM.
    - View many more results by following the link, but tha's the jist of it.
- InnoDB benchmarks as run by Basho.
    - LevelDB showed a higher throughput than InnoDB and a similar or lower latency than InnoDB.
    - LevelDB may become a preferred choice for Riak users whose data set has massive numbers of keys and therefore is a poor match with Bitcask's model.
    - Before LevelDB can be a first-class storage engine under Riak it must be portable to all of the same platforms that Riak is supported on.
- LEVELDB VS KYOTO CABINET MY FINDINGS. Ecstortive says wait a minute here, Kyoto is actually faster.
- A good sign of adoption, language bindings are being built: Java, Tie::LevelDB on CPAN
- Comparing LevelDB and Bitcask: LevelDB is a persistent ordered map; bitcask is a persistent hash table (no ordered iteration). Bitcask stores a fixed size record in memory for every key. So for databases with large number of keys, it may use too much memory for some applications. Bitcask can guarantee at most one disk seek per lookup I think. LevelDB may have to do a small handful of disk seeks. To clarify, leveldb stores data in a sequence of levels. Each level stores approximately ten times as much data as the level before it. A read needs one disk seek per level. So if 10% of the db fits in memory, leveldb will need to do one seek (for the last level since all of the earlier levels should end up cached in the OS buffer cache). If 1%

fits in memory, leveldb will need two seeks. Bitcask is a combination of Erlang and C.

- Writes can be lost, but that doesn't trash the data files: Leveldb never writes in place: it always appends to a log file, or merges existing files together to produce new ones. So an OS crash will cause a partially written log record (or a few partially written log records). Leveldb recovery code uses checksums to detect this and will skip the incomplete records.
- LevelDB is being used as the back-end for IndexedDB in Chrome. For designing how to map secondary indices into LevelDB key/values, look at how the IndexedDB support within Chrome is implemented.
- In case of a crash partial writes are ignored.
- Possible scalability issues:
    - LevelDB keeps a separate file for every couple of MB of data, and these are all in one directory. Depending on the underlying file system, this might start causing trouble at some point.
    - Scalability is more limited by the frequency of reads and writes that are being done, rather than the number of bytes in the system.
- Transactions are not supported. Writes (including batches) are atomic. Consistency is up to you. There is limited isolation support. Durability is a configurable option. Full blown ACID transactions require a layer on top of LevelDB (see WebKit's IndexedDB).
- Michi Mutsuzaki compared LevelDB to MySQL as a key-value store. LevelDB had better overall insert throughput, but it was less stable (high variation in throughput and latency) than mysql. There was no significant performance difference for 80% read / 20% update workload.
- LevelDB hasn't been tuned for lots of concurrent readers and

writers. Possible future enhancements:

1. Do not hold the mutex while the writer is appending to the log (allow concurrent readers to proceed)
2. Implement group commit (so concurrent writers have their writes grouped together).

Related Articles

- On Hacker News
- The Log-Structured Merge-Tree (LSM-Tree) by Patrick O'Neil
- Cache Conscious Indexing for Decision-Support in Main Memory
- LevelDB Compared to nessDB
- Bigtable: A Distributed Storage System for Structured Data
- Hot Trend: Move Behavior To Data For A New Interactive Application Architecture

---