Cockroach LABS

SYSTEMS

Why we built CockroachDB on top of RocksDB

Written by Arjun Narayan and Peter Mattis on January 17, 2019

If, on a final exam at a database class, you asked students whether to build a database on a log-structured merge tree (LSM) or a BTree-based storage engine, 90% of your students would probably respond that the decision hinges on your workload. “LSMs are for write-heavy workloads and BTrees are for read-heavy workloads”, the conscientious ones would write. If you surveyed most NewSQL databases today, most of them are built on top of an LSM, namely, RocksDB. You might thus conclude that this is because modern applications have shifted to more write-heavy workloads. You would be incorrect.

The main motivation behind RocksDB adoption has nothing to do with its choice of LSM data structure. In our case, the compelling drivers were its rich feature set which turns out to be necessary for a complex product like a distributed database. At Cockroach Labs we use RocksDB as our storage engine and depend on a lot of features that are not

Cockroach LABS

What is a storage engine?

A storage engine's job is to write things to disk on a single node. For many databases — which may be single node databases themselves — this constitutes a large part of the engineering effort, and so is pretty intricately tied to the engineering effort of the database building itself.

However, in a distributed setting, the distribution, replication, and transaction coordination parts are an added engineering complexity, that at Cockroach Labs we began to look to a mature storage engine product we could build on, rather than building one from scratch.

So what exactly is a storage engine? A simple first answer, considering the big asks for any database – Atomicity, Consistency, Isolation, and Durability, or ACID – is that the storage engine's responsibility is **A**tomicity and **D**urability. This frees up the higher layers of the database to focus on the distributed coordination required to get strong **I**solation guarantees like serializability, and providing the **C**onsistency primitives needed to ensure data integrity.

Beyond atomicity and durability though, a storage engine has another big job: performance. The performance characteristics of a storage engine determine to a large extent the ceiling on the performance of the entire database. For example: almost every storage engine has a write-ahead log for quickly making a write durable, but which will later get moved to the main indexed data structure. This is a performance optimization that is pretty vital, but also tricky to get right, and involves delicate performance tradeoffs, as committing writes to the write-ahead-log while maintaining high concurrency is tricky.

Storage engines also have to provide a defined isolation model (e.g. that your reads will reflect writes that are still in the write-ahead log and will be as of some single point-in-time snapshot), while supporting many concurrent operations. While that isolation model might be simpler than the overall isolation provided by the database (for instance, RocksDB provides snapshot isolation, while CockroachDB does extra

If this sounds like a lot of the work that goes into building a database, it is! For instance, Postgres doesn't really have a defined "storage engine" – it's all one monolithic system. But for a distributed database, a single node's storage engine is a smaller part of the larger distributed system, so let's take a deeper dive into understanding how we use RocksDB in CockroachDB, paying attention to the lesser-known features that we use.

A RocksDB primer

RocksDB is a single-node key-value storage engine. The design is based on log-structured merge trees (LSMs). RocksDB is a fork of an earlier Google project called LevelDB, which was an embedded key-value store inspired by the low-level storage engine used by BigTable. RocksDB has since gone on to become a much more robust and feature complete storage engine, but the basic structure is the same as LevelDB and many other LSM-based storage engines.

In RocksDB, keys and values are stored as sorted strings in files called *SSTables*. These SSTables are arranged in several *levels*. Within a single level, SSTables are non-overlapping: one SSTable might contain keys covering the range `[a,b)` , the next `[b,d)` , and so on. The key-space does overlap between levels: if you have two levels, the first might have two SSTables (covering the ranges above), but the second level might have a single SSTable over the keyspace `[a,e)` . Looking for the key `aardvark` requires looking in two SSTables: the `[a,b)` SSTable in Level 1, and the `[a,e)` SSTable in Level 2. Each SSTable is internally sorted (as in the name), so lookups within an SSTable take `log(n)` time. SSTables store their keys in *blocks*, and have an internal *index*, so even though a single SSTable may be very large (gigabytes in size), only the index and the relevant block needs to be loaded into memory.

The levels are structured roughly so that each level is in total 10x as large as the level above it. New keys arrive at the highest layer, and as that level gets larger and larger and hits a threshold, some SSTables at that level get *compacted* into fewer (but larger) SSTables one level lower. The precise details of when to compact (and how) greatly affect performance; Leveled Compaction is one compaction strategy, but for an

Above the on-disk levels is an in-memory *memtable*. The memtable is a sorted in-memory data structure (we use a concurrent skiplist, although RocksDB has several options), which makes reads cheap, but is persisted as an *unsorted* Write-Ahead-Log (WAL). If a node crashes, on startup, the durable WAL is read back and the memtable is reconstructed. As this data-structure grows with more writes, it needs to be flushed to disk. This eviction can be a critical bottleneck during sustained write throughput. In order to make memtable flushes cheap, L0 is special: its SSTables are allowed to overlap. This, however, makes L0 a critical bottleneck for compactions and read performance — it is usually compacted in large chunks into L1, and every table in L0 increases read amplification.

Thus, as you can see, writes create *deferred* write amplification, in the form of eventual compactions that will eventually push the keys down the hierarchy of levels. Bursty write workloads can accommodate a lot of writes. Sustained writes will require dedicating some portion of IO bandwidth to performing some compactions concurrently (a decent amount of the motivation for the initial RocksDB project was in using multicore concurrency to solve this problem more efficiently than in LevelDB).

Translating higher level SQL operations into K and V operations

CockroachDB is a distributed SQL database. These SQL operations are translated down into key and value operations over a single logical keyspace. This logical keyspace is sharded into physical ‘ranges’ of keys, and each range is replicated across three (or more) Cockroach nodes. Given this structure, a given SQL operation gets turned into a set of key value operations, which are spread across multiple machines. At a given machine, these KV operations thus need to be performed on the underlying storage engine.

This API is relatively simple, since RocksDB also provides a key-value interface. But what exactly do we mean by ‘key-value interface’? This lack of standardization hides a lot of subtle detail: this interface is more than just `put`, `get`, and `delete` operations on t' keys. RocksDB also supports scans over a range `[start, end)`. Also consider other

Cockroach LABS

critical, as otherwise some SQL operations can become very slow. Let's cover why these are so critical to a distributed database like CockroachDB.

Fast scans

One surprising part of engineering CockroachDB is the realization that scans are more frequent than you would think! Many academic papers use put/get operations for testing storage engine performance, such as by using YCSB as a storage engine benchmark. However, in CockroachDB, `put/scan` are the two most dominant operations because of the higher level guarantees we provide as a serializable SQL database. Consider multi-version concurrency control (MVCC) – Cockroach stores multiple values of a given key, and also stores the timestamp at which each key was written. Transactions happen at a particular timestamp, and are guaranteed to read the *latest* value as of that timestamp. Thus, what may appear to be a `GET` operation at the database level (e.g. `SELECT * from tablename WHERE primarykeycol = 123` translates into a `GET` for the value of the key that stores that row), turns into a `SCAN` at the storage engine level (`SCAN` for the *newest* value of that key). Thus each CockroachDB key is annotated with a timestamp to create the equivalent RocksDB key. And updates to a key create additional RocksDB keys.

The use of MVCC causes additional complications. Many key-value storage engines have fast `GET` operations, but slower `SCAN` operations. Consider any log-structured merge tree (LSM) implementation: a particular key can be in any of the levels of the LSM. This means that every `GET` operation has a *read amplification factor* (RAF) of the number of levels (One logical read = RAF disk reads). In order to alleviate this `log(n)` multiple, storage engines typically use bloom filters on SSTs. Bloom filters are probabilistic data structures that are small enough to keep in memory. A bloom filter answers the question: "is a given key present in this level?" with either a "no" or "maybe" (if it's a "maybe", you have to perform the read to find out the answer, but a "no" means you can skip reading).

Unfortunately, bloom filters are per-key. A `SCAN` involves a potentially infinite keyspace between two endpoints, so the bloom filter cannot be used to rule out levels to sc
That is unless you can pre-process keys when constructing your the bloom filter.

that prefix can benefit from using the bloom filter. Without this, a storage engine will have to scan every level on every logical `GET` operation, a huge performance hit. Prefix bloom filters, like RocksDB provides, are thus table stakes for an MVCC database like CockroachDB.

RocksDB Snapshots

Cockroach's distributed replication means that occasionally, a new node needs to be brought up to speed with a copy of some data. This requires that a large chunk of the keyspace be scanned, and sent over the network to the new node. Depending on the size of the data and the speed of the network, this operation can take a decent amount of time (from a few seconds to tens of seconds).

If it takes a while, Cockroach has two options: do the scan over a long period of time, or do the entire scan and hold the data separately until the transmission is complete. Most storage engines, including RocksDB, provide the feature that any given scan operation is done over a consistent 'snapshot' of the database — writes done after the scan has started will not be reflected in the scan operation. This isolation guarantee at the storage level is a useful building block for the database to build on top of. It also comes in use in sending these snapshots for replication, but the challenge comes in providing the snapshot functionality without using too many resources. In particular, reading a snapshot pins memtables during the read operation, which slows down incoming writes. Naively pinning memtables can be expensive, as is the alternative of reading out the snapshot to free up the storage engine, only to then have to hold those keys in memory at a higher level until the transmission completes.

This problem is exacerbated by the fact that CockroachDB shards the keyspace into many ranges — each up to 64mb. This means that when a new node comes up, it is likely to cause many snapshots to be created and transmitted across the entire cluster before it fills up to parity with the other nodes. And all of this while the cluster itself has to continue normal operation.

RocksDB provides an alternate middle ground — the explicit snapshot. Explicit snapshots do not pin memtables. Instead, they are a sort of placeholder that informs not consume additional resources (apart from preventing some compactions that might make your storage layout more efficient). When you're ready to iterate over the data, you create an implicit snapshot (which might have to pin memtables for the duration of the iterator), but this way, you don't have to hold on to an expensive implicit snapshot for a long time.

Blitzing through more RocksDB features

We use a lot of RocksDB features in CockroachDB, and covering all of them exhaustively would take a lot more pages! So here's a short preview into many RocksDB features that we use:

SSTable ingestion

During restoration of a backup, we want to ingest files that contain keys that span a portion of the keyspace which we can guarantee is empty when we begin the ingestion. Using the normal write path is wasteful — the normal write path involves writing to high levels of the LSM, and then compacting down, which involves a lot of write amplification. If we know this keyspace is empty, we can simply pre-construct SSTables at a low level that are guaranteed to not overlap the other SSTables. This greatly improves restore throughput. This is a fairly critical feature for CockroachDB, and thus ingesting of SSTables into RocksDB is a key feature.

Custom key comparators

The MVCC timestamp suffix that we tack onto keys could be encoded in such a way that lexicographical comparison (byte for byte) is equivalent to logical comparison. But such an encoding is slower to encode and decode than one which does not compare lexicographically. RocksDB comes to the rescue again by allowing a custom comparator to be defined for the key. This enables an encoding of the MVCC timestamp that is quick to encode and decode, while also allowing the correct ordering to be maintained (timestamps are sorted in *descending* order since we want the latest version of the key

Cockroach LABS

Range Deletion Tombstones

Efficiently deleting an entire range of keys is required, otherwise operations like `DROP TABLEs` or simply moving a chunk of data between nodes can block for really long periods of time. (Non-engineers: in a computer, a move is always implemented as a copy followed by a delete). RocksDB achieves this by performing delete range operations by writing a *range deletion tombstone*. When reading a key, if a range tombstone that covers that key is read at a higher level than a concrete value for that key, then the key is considered deleted. The actual deletions in the SSTables are performed during compaction.

Backwards iteration

Backwards iteration makes queries like `SELECT * FROM TABLE ORDER BY key DESC LIMIT 100` efficient, even if the index on `key` is ordered ascending. Some storage engines do not provide the ability to iterate backwards, which makes those queries inefficient. It's worth noting that backwards iteration is always going to be more expensive than forward iteration without changes to the underlying data layout. RocksDB abstracts away the complexities from the higher levels of the system. It is worth noting that there are opportunities for improved performance in this area.

Indexed Batches

Batches are the unit of atomic write operations (containing set, delete, or delete range operations). The basic batch is write-only. RocksDB also supports an "indexed batch" that allows reads from the batch. In a distributed database, some writes can take longer, as CockroachDB waits for remote operations to commit. However, other operations *in the same transaction* need to be able to read the pending writes. To support this, we rely on the storage engine to provide a mechanism for batching a set of updates that can be applied atomically, while also providing a means to read a

Encryption Support

For an upcoming feature — encryption at rest — we rely on RocksDB's modular support for encrypting SSTables. This does a lot of the heavy lifting of keeping the data encrypted, so that we can focus on key management, rotation, and the user-facing parts of supporting encryption at rest.

RocksDB at CockroachDB today

Today RocksDB is deeply embedded in Cockroach's architecture. Other storage engines that don't have the above features would require significant re-engineering on our part in order to adopt them and even then, would probably result in performance degradation. Promised performance increases on raw key/value access speed are quite likely to disappear once all these considerations are taken into account.

That said, RocksDB is not all roses. After all, we've only gotten this performance after expending considerable effort in engineering RocksDB for our needs! There are also downsides to having a performance critical part of our codebase in C++ while the rest of the system is in Go. Crossing the CGo barrier is delicate – and involves a 70ns overhead per call. That sounds fast (a nanosecond is a billionth of a second), but we perform a lot of RocksDB calls. Minimizing the number of CGo crossings has a measurable impact! We now construct our entire batch of RocksDB operations (e.g. a set of PUTs) in Go, and then transfer them in a single CGo call for efficiency. We also incur performance penalties in having to copy values from C allocated memory into Go allocated memory. A Go-native storage engine could provide us many performance benefits, as well as streamlining our codebase, though simply using an existing Go-native storage engine is a bit of a non-starter given the requirements above (we're not aware of any which provide all of the functionality we need). Given the choice of implementing all of these delicate performance-critical features or engineering around minimizing CGo overhead, we've found the latter manageable so far, but we're keeping an eye on when this calculation changes.

and checkpoints, persistent caches, and transactions… Perhaps we'll find performance reasons to include them in upcoming versions of CockroachDB!

If building distributed systems is your cup of tea, check out our open positions here.

engineering rocksdb