medium.com

Bitcask — a log-structured fast KV store - Arpit - Medium

Arpit

8-10 minutes

Bitcask is one of the most efficient embedded Key-Value (KV) Databases designed to handle production-grade traffic. The paper that introduced Bitcask to the world says it is a *Log-StructuredHash Table for Fast Key/Value Data* which, in a simpler language, means that the data will be written sequentially to an append-only log file and there will be pointers for each `key` pointing to the `position` of its log entry. Building a KV store off the append-only log files seems like a really weird design choice, but Bitcask does not only make it efficient but it also gives a really high Read-Write throughput.

Bitcask was introduced as the backend for a distributed database named Riak in which each node used to run one instance of Bitcask to hold the data that it was responsible for. In this essay, we take a detailed look into Bitcask, its design, and find the secret sauce that makes it so performant.

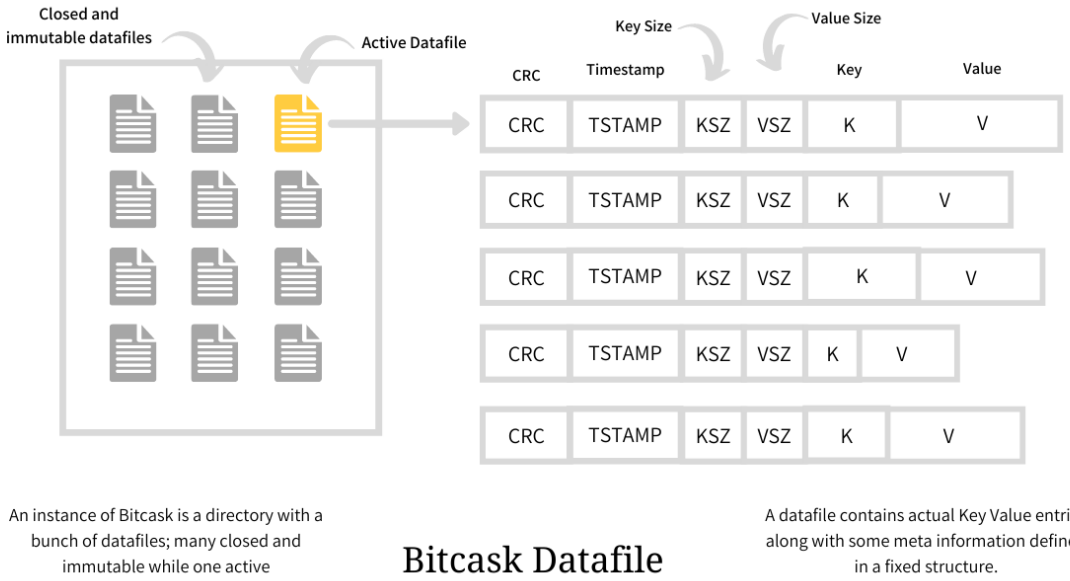
Design of Bitcask

Bitcask uses a lot of principles from log-structured file systems and

draws inspiration from a number of designs that involve log file merging, for example — merging in LSM Trees. It essentially is just a directory of append-only (log) files with a fixed structure and an in-memory index holding the keys mapped to a bunch of information necessary for point lookups — referring to the entry in the datafile.

Datafiles

Datafiles are append-only log files that hold the KV pairs along with some meta-information. A single Bitcask instance could have many data files, out of which just one will be active and opened for writing, while the others are considered immutable and are only used for reads.

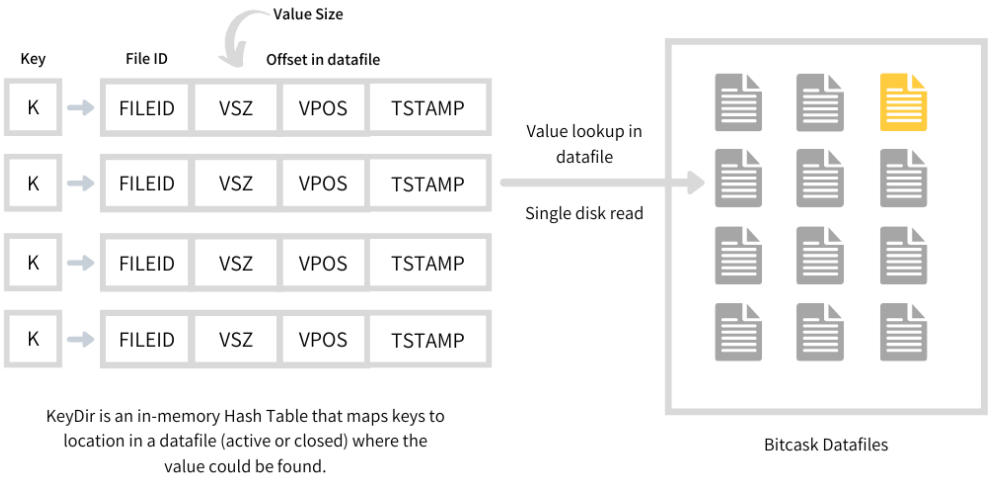


Each entry in the datafile has a fixed structure illustrated above and it stores `crc`, `timestamp`, `key_size`, `value_size`, actual `key`, and the actual `value`. All the write operations - create, update and delete - made on the engine translates into entries in this active datafile. When this active datafile meets a size threshold, it is closed and a new active datafile is created; and as stated earlier,

when closed (intentionally or unintentionally), the datafile is considered immutable and is never opened for writing again.

KeyDir

KeyDir is an in-memory hash table that stores all the keys present in the Bitcask instance and maps it to the offset in the datafile where the log entry (value) resides; thus facilitating the point lookups. The mapped value in the Hash Table is a structure that holds `file_id`, `offset`, and some meta-information like `timestamp`, as illustrated below.



Bitcask KeyDir

Operations on Bitcask

Now that we have seen the overall design and components of Bitcask, we can jump into exploring the operations that it supports and details of their implementations.

Putting a new Key Value

When a new KV pair is submitted to be stored in the Bitcask, the

engine first appends it to the active datafile and then creates a new entry in the KeyDir specifying the offset and file where the value is stored. Both of these actions are performed atomically which means either the entry is made in both the structures or none.

Putting a new Key-Value pair requires just one atomic operation encapsulating one disk write and a few in-memory access and updates. Since the active datafile is an append-only file, the disk write operation does not have to perform any disk seek whatsoever making the write operate at an optimum rate providing a high write throughput.

Updating an existing Key Value

This KV store does not support partial update, out of the box, but it does support full value replacement. Hence the update operation is very similar to putting a new KV pair, the only change being instead of creating an entry in KeyDir, the existing entry is updated with the new position in, possibly, the new datafile.

The entry corresponding to the old value is now dangling and will be garbage collected explicitly during merging and compaction.

Deleting a Key

Deleting a key is a special operation where the engine atomically appends a new entry in the active datafile with value equalling a tombstone value, denoting deletion, and deleting the entry from the in-memory KeyDir. The tombstone value is chosen as something very unique so that it does not interfere with the existing value space.

Delete operation, just like the update operation, is very lightweight and requires a disk write and an in-memory update. In delete operation as well, the older entries corresponding to the deleted keys are left dangling and will be garbage collected explicitly during merging and compaction.

Reading a Key-Value

Reading a KV pair from the store requires the engine to first find the datafile and the offset within it for the given key; which is done using the KeyDir. Once that information is available the engine then performs one disk read from the corresponding datafile at the offset to retrieve the log entry. The correctness of the value retrieved is checked against the CRC stored and the value is then returned to the client.

The operation is inherently fast as it requires just one disk read and a few in-memory accesses, but it could be made faster using Filesystem read-ahead cache.

Merge and Compaction

As we have seen during Update and Delete operations the old entries corresponding to the key remain untouched and dangling and this leads to Bitcask consuming a lot of disk space. In order to make things efficient for the disk utilization the engine once a while compacts the older closed datafiles into one or many merged files having the same structure as the existing datafiles.

The merge process iterates over all the immutable files in the Bitcask and produces a set of datafiles having only *live* and *latest* versions of each present key. This way the unused and non-

existent keys are ignored from the newer datafiles saving a bunch of disk space. Since the record now exists in a different merged datafile and at a new offset, its entry in KeyDir needs an atomic updation.

Performant bootup

If the Bitcask crashes and needs a boot-up, it will have to read all the datafiles and build a new KeyDir. Merging and compaction here do help as it reduces the need to read data that is eventually going to be evicted. But there is another operation that could help in making the boot times faster.

For every datafile a *hint* file is created which holds everything in the datafile except the value i.e. it holds the key and its meta-information. This *hint* file, hence, is just a file containing all the keys from the corresponding datafile. This *hint* file is very small in size and hence by reading this file the engine could quickly create the entire KeyDir and complete the bootup process faster.

Strengths and Weaknesses of Bitcask

Strengths

- Low latency for read and write operations
- High Write Throughput
- Single disk seek to retrieve any value
- Predictable lookup and insert performance
- Crash recovery is fast and bounded
- Backing up is easy — Just copy the directory would suffice

Weaknesses

The KeyDir holds all the keys in memory at all times and this adds a huge constraint on the system that it needs to have enough memory to contain the entire keyspace along with other essentials like Filesystem buffers. Thus the limiting factor for a Bitcask is the limited RAM available to hold the KeyDir.

Although this weakness sees a major one but the solution to this is fairly simple. We can typically shard the keys and scale it horizontally without losing much of the basic operations like Create, Read, Update, and Delete.

References

- Bitcask Paper
- Bitcask — Wikipedia
- Riak's Bitcask — High Scalability
- Implementation of the Bitcask storage model-merge and hint files

Other articles that you might like

- Consistent Hashing
- Phi φ Accrual Failure Detection
- Copy-on-Write Semantics
- What makes MySQL LRU cache scan resistant

If you liked what you read, consider subscribing to my weekly newsletter at arpitbhayani.me/newsletter where, once a week, I write an essay about programming languages internals, or a deep

dive on some super-clever algorithm, or just a few tips on building highly scalable distributed systems.

You can always find me browsing through twitter @arpit_bhayani.