Bitcask

A Log-Structured Hash Table for Fast Key/Value Data

Justin Sheehy David Smith
with inspiration from Eric Brewer

Basho Technologies

justin@basho.com,dizzyd@basho.com

The origin of Bitcask is tied to the history of the Riak distributed database. In a Riak key/value cluster, each node uses pluggable local storage; nearly anything k/v-shaped can be used as the per-host storage engine. This pluggability allowed progress on Riak to be parallelized such that storage engines could be improved and tested without impact on the rest of the codebase.

Many such local key/value stores already exist, including but not limited to Berkeley DB, Tokyo Cabinet, and Innostore. There are many goals we sought when evaluating such storage engines, including:

- low latency per item read or written
- high throughput, especially when writing an incoming stream of random items
- ability to handle datasets much larger than RAM w/o degradation
- crash friendliness, both in terms of fast recovery and not losing data
- ease of backup and restore
- a relatively simple, understandable (and thus supportable) code structure and data format
- predictable behavior under heavy access load or large volume
- a license that allowed for easy default use in Riak

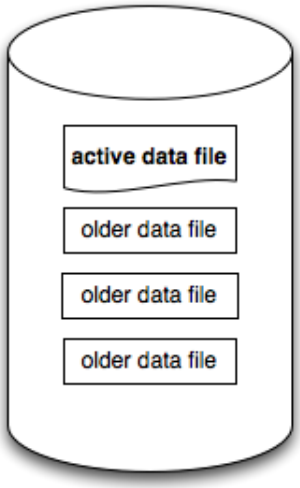
Achieving some of these is easy. Achieving them all is less so.

None of the local key/value storage systems available (including but not limited to those written by the authors) were ideal with regard to all of the above goals. We were discussing this issue with Eric Brewer when he had a key insight about hash table log merging: that doing so could potentially be made as fast or faster than LSM-trees. LOG STRUCTURED MERGE TREES.

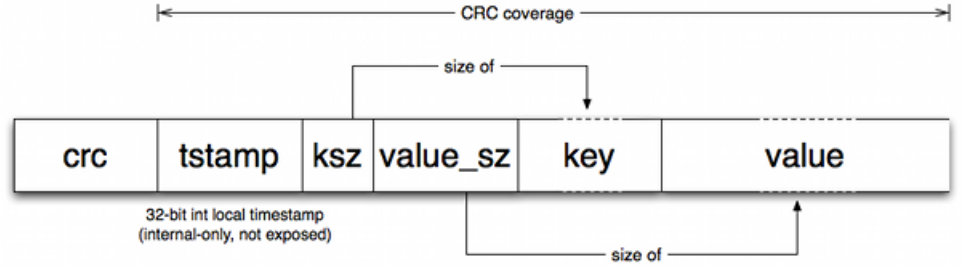
This led us to explore some of the techniques used in the log-structured file systems first developed in the 1980s and 1990s in a new light. That exploration led to the development of Bitcask, a storage system that meets all of the above goals very well. While Bitcask was originally developed with a goal of being used under Riak, it was built to be generic and can serve as a local key/value store for other applications as well.

The model we ended up going with is conceptually very simple. A Bitcask instance is a directory, and we enforce that only one operating system process will open that Bitcask for writing at a given time. You can think of that process effectively as the ”database server”. At any moment, one file is ”active” in that directory for writing by the server. When that file meets a size threshold it will be closed and a new active file will be created. Once a file is closed, either purposefully or due to server exit, it is considered immutable and will never be opened for writing again.

a bitcask on disk



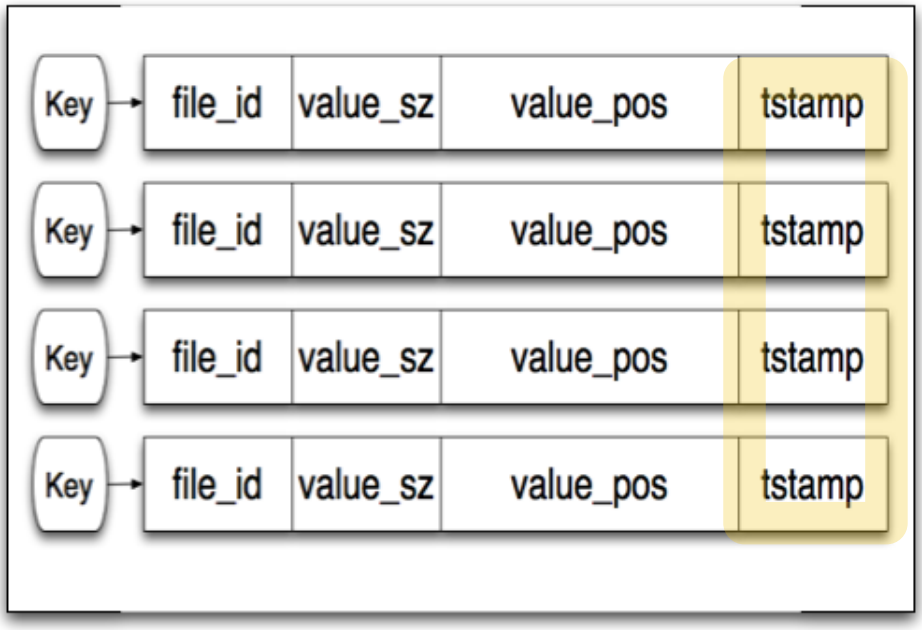
The active file is only written by appending, which means that sequential writes do not require disk seeking. The format that is written for each key/value entry is simple:



With each write, a new entry is appeneded to the active file. Note that deletion is simply a write of a special tombstone value, which will be removed on the next merge. Thus, a Bitcask data file is nothing more than a linear sequence of these entries:

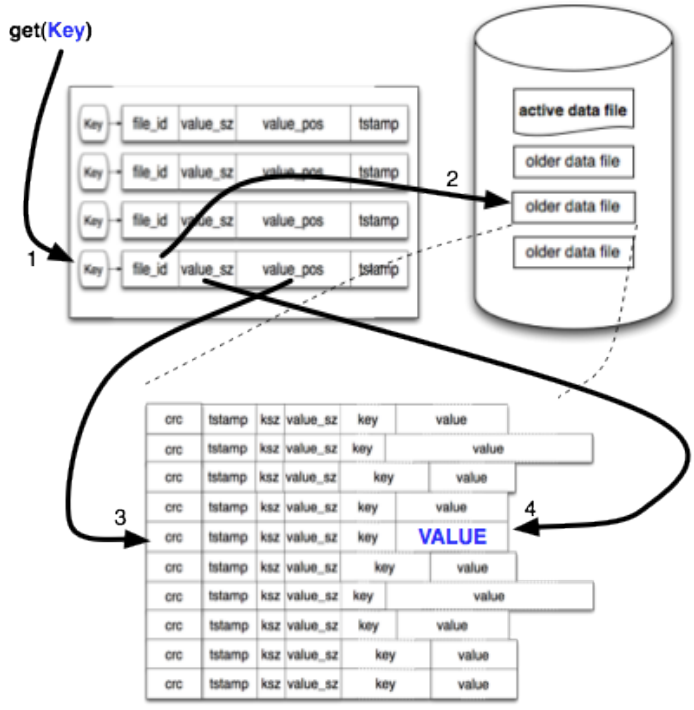
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value
crc	tstamp	ksz	value_sz	key	value

After the append completes, an in-memory structure called a ”keydir” is updated. A keydir is simply a hash table that maps every key in a Bitcask to a fixed-size structure giving the file, offset, and size of the most recently written entry for that key.



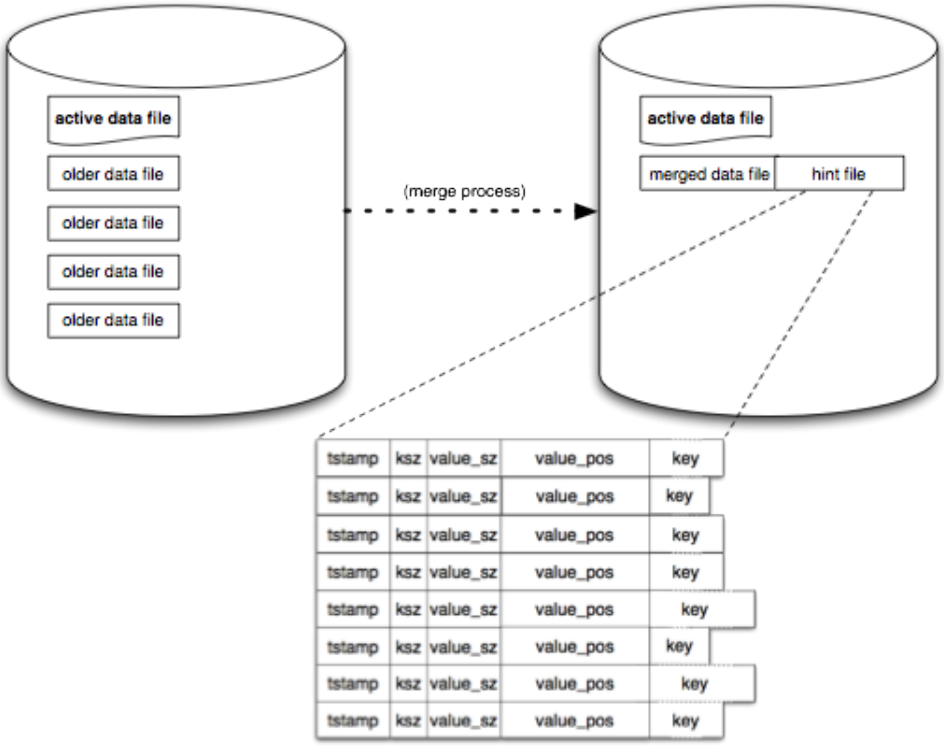
When a write occurs, the keydir is atomically updated with the location of the newest data. The old data is still present on disk, but any new reads will use the latest version available in the keydir. As we’ll see later, the merge process will eventually remove the old value.

Reading a value is simple, and doesn’t ever require more than a single disk seek. We look up the key in our keydir, and from there we read the data using the file_id, position, and size that are returned from that lookup. In many cases, the operating system’s filesystem read-ahead cache makes this a much faster operation than would be otherwise expected.



This simple model may use up a lot of space over time, since we just write out new values without touching the old ones. A process for compaction that we refer to as ”merging” solves this. The merge process iterates over all non-active (i.e. immutable) files in a Bitcask and produces as output a set of data files containing only the ”live” or latest versions of each present key.

When this is done we also create a ”hint file” next to each data file. These are essentially like the data files but instead of the values they contain the position and size of the values within the corresponding data file.



When a Bitcask is opened by an Erlang process, it checks to see if there is already another Erlang process in the same VM that is using that Bitcask. If so, it will share the keydir with that process. If not, it scans all of the data files in a directory in order to build a new keydir. For any data file that has a hint file, that will be scanned instead for a much quicker startup time.

These basic operations are the essence of the bitcask system. Obviously, we've not tried to expose every detail of operations in this document; our goal here is to help you understand the general mechanisms of Bitcask. Some additional notes on a couple of areas we breezed past are probably in order:

- We mentioned that we rely on the operating system's filesystem cache for read performance. We have discussed adding a bitcask-internal read cache as well, but given how much mileage we get for free right now it's unclear how much that will pay off.
- We will present benchmarks against various API-similar local storage systems very soon. However, our initial goal with Bitcask was not to be the fastest storage engine but rather to get ”enough” speed and also high quality and simplicity of code, design, and file format. That said, in our initial simple benchmarking we have seen Bitcask handily outperform other fast storage systems for many scenarios.
- Some of the hardest implementation details are also the least interesting to most outsiders, so we haven't included in this short document a description of (e.g.) the internal keydir locking scheme.
- Bitcask does not perform any compression of data, as the cost/benefit of doing so is very application-dependent.

And let's look at the goals we had when we set out:

- low latency per item read or written

  Bitcask is fast. We plan on doing more thorough benchmarks soon, but with sub-millisecond typical median latency (and quite adequate higher percentiles) in our early tests we are confident that it can be made to meet our speed goals.

- high throughput, especially when writing an incoming stream of random items

  In early tests on a laptop with slow disks, we have seen throughput of 5000-6000 writes per second.

- ability to handle datasets much larger than RAM w/o degradation

  The tests mentioned above used a dataset of more than 10×RAM on the system in question, and showed no sign of changed behavior at that point. This is consistent with our expectations given the design of Bitcask.

- crash friendliness, both in terms of fast recovery and not losing data

  As the data files and the commit log are the same thing in Bitcask, recovery is trivial with no need for "replay." The hint files can be used to make the startup process speedy.

- ease of backup and restore

  Since the files are immutable after rotation, backup can use whatever system-level mechanism is preferred by the operator with ease. Restoration requires nothing more than placing the data files in the desired directory.

- a relatively simple, understandable (and thus supportable) code structure and data format

  Bitcask is conceptually simple, clean code, and the data files are very easy to understand and manage. We feel very comfortable supporting a system resting atop Bitcask.

- predictable behavior under heavy access load or large volume

  Under heavy access load we've already seen Bitcask do well. So far it has only seen double-digit gigabyte volumes, but we'll be testing it with more soon. The shape of Bitcask is such that we do not expect it to perform too differently under larger volume, with the one predictable exception that the keydir structure grows by a small amount with the number of keys and must fit entirely in RAM. This limitation is minor in practice, as even with many millions of keys the current implementation uses well under a GB of memory.

In summary, given this specific set of goals, Bitcask suits our needs better than anything else we had available.

The API is quite simple:

bitcask:open(DirectoryName, Opts) → BitCaskHandle | {error, any()}	Open a new or existing Bitcask datastore with additional options. Valid options include read_write (if this process is going to be a writer and not just a reader) and sync_on_put (if this writer would prefer to sync the write file after every write operation). The directory must be readable and writable by this process, and only one process may open a Bitcask with read_write at a time.
bitcask:open(DirectoryName) → BitCaskHandle | {error, any()}	Open a new or existing Bitcask datastore for read-only access. The directory and all files in it must be readable by this process.
bitcask:get(BitCaskHandle, Key) → not_found | {ok, Value}	Retrieve a value by key from a Bitcask datastore.
bitcask:put(BitCaskHandle, Key, Value) → ok | {error, any()}	Store a key and value in a Bitcask datastore.
bitcask:delete(BitCaskHandle, Key) → ok | {error, any()}	Delete a key from a Bitcask datastore.
bitcask:list_keys(BitCaskHandle) → [Key] | {error, any()}	List all keys in a Bitcask datastore.
bitcask:fold(BitCaskHandle,Fun,Acc0) → Acc	Fold over all K/V pairs in a Bitcask datastore. Fun is expected to be of the form: F(K,V,Acc0) → Acc.
bitcask:merge(DirectoryName) → ok | {error, any()}	Merge several data files within a Bitcask datastore into a more compact form. Also, produce hintfiles for faster startup.
bitcask:sync(BitCaskHandle) → ok	Force any writes to sync to disk.
bitcask:close(BitCaskHandle) → ok	Close a Bitcask data store and flush all pending writes (if any) to disk.

As you can see, Bitcask is quite simple to use. Enjoy!

CRC - CYCLIC REDUNDANCY CHECK (ERROR DETECTING CODE).

READ-AHEAD CACHE - PREFETCHING FILE / LOADING FILE INTO THE PAGE $ CACHE $

MACOS 10.14.4, 18.5.0 - PAGE SIZE 4KB.

BASIC BITCASK OPERATION