High Performance Key-Value Stores

Tobias Pristupin

December, 2023

Contents

1 Project Overview

This project was a semester long independent study with the goal of investigating the techniques and tradeoffs involved with developing a high performance persisting key-value store.

The goal of this project was to follow Martin Kleppman's book *Designing Data Intensive Applications* [5] and implement the Log Database and SSTable Database, with some modifications. We then collect data on their performance. The first section of this paper is dedicated to the Log Database, and the second to the SSTable Database. Finally, we present some benchmarks. All the code for this project is in C++ and can be found at https://github.com/TobiPristupin/databases.

2 The Log Database

The Log Database stands out as a straightforward database implementation, yet its performance is unexpectedly robust. The fundamental idea is to store all data in an append-only immutable [1] log file.

2.1 Reads and Writes

Every time a write is performed, we append the key-value pair to the log file. When reading, there is no need to scan the entire log file. Instead, we maintain an in-memory mapping of keys to their respective offset in the log file. Reads require a $\sim \mathcal{O}(1)$ read of the offset table, and a corresponding seek and read in the log file. It is often the case that the log file is already cached in-memory by the Operating System, which would render the read a completely in-memory operation. Writes are similarly performant, as we enforce that our log file is append-only, which makes writing fast. Again, if the log file is cached — which will often be the case for the end segment of the log file — the write will be performed completely in memory.

Both reads and writes are $\mathcal{O}(1)$. Delegating the work of scheduling I/O operations to the Operating System simplifies the implementation and provides optimal performance.

2.2 Persistence and Deletes

By backing all data in the log file, the database automatically becomes persistent. When initializing the database, it performs a one-time scan of the log file, populating the offset table. Since every operation is persisted immediately, we become resilient to crashes also (assuming operations are atomic).

This does introduce a small challenge in the case of deletions. Suppose we write a key-value pair to our log file, and then delete it by removing it from our offset table. The next time the database restarts, it will scan the log file and populate the deleted key in the offset table. To remediate this, we must add a *tombstone* to the log file, a special entry that records a key deletion.

2.3 Encoding the Log File

A simple implementation can make use of a text file with a format such as csv. To achieve maximum space efficiency, our implementation uses a custom binary format. The log file is composed of a series of entries. Each entry has a header composed of three 4 byte integers. The first two correspond to the lengths of the key and the value, respectively. The third integer contains information on the type of the value. The database supports C++ types `int`,`double`,`float`,`bool`,`std::string`, and the key is fixed as a string. The header is followed by the key and the value. A value length of 0 corresponds to a tombstone entry.

[1] Appending is allowed, but modifying previous values is not

3 SSTable Database

The SSTable database, short for Sorted String Table, was initially proposed by Patrick O'Neil et. al under the name LSM-Tree, and was popularized by Google [4], who introduced the term SSTable. It is currently used in Google's BigTable [1], as well as other commercially available databases, such as Apache Cassandra [2] and ScyllaDB [3].

The database is split into two components: an in-memory cache which we will refer to as *memtable*, and a series of files on disk which we will refer to as *SSFiles*. An SSFile is an immutable read-only file which contains key-value pairs sorted by key. When the memtable exceeds a certain threshold, it is flushed onto a new SSFile. Writes and deletes go straight to the cache, whereas reads go first to the memtable, and if the key is not found, the database will then consult all the SSFiles. The most recent SSFile is read first, to obtain the most updated value.

3.1 Flushing the Memtable

Flushing can be an expensive operation, because we require the SSFile to be sorted. To alleviate this, we also maintain the memtable sorted, using a data structure such as a Binary Search Tree or Red-Black tree. This does require us to forgo a $\mathcal{O}(1)$ cache implementation for an $\mathcal{O}(\log n)$ one, but it speeds up flushes, as one can simply traverse the data structure in order when flushing.

Deciding the threshold to trigger a flush is also something to consider. A value too large could risk having the memtable not fit in memory, as well as flushes becoming too costly. A value too small might not take full advantage of the memtable.

3.2 Design of the SSFile

An SSFile must support a fast read. Writes and deletes are unnecessary, as those are handled by the memtable. We have the advantage that SSFiles are immutable, so we know all the data at time of creation. We want to store the keys in sorted order, so we can use Binary Search to find keys.

The underlying issue with binary search on an SSFile is that both keys and values have variable sizes, which complicates random access. A simple encoding could be to use a text file with csv formatting, where each line contains one entry. However, performing binary search on such files would require one to load all lines into memory. This can be optimized by performing the binary search in chunks, or by using a sparse index that maps a tiny subset of keys to their file offsets. We propose an encoding scheme that does not require loading all lines to memory, and can be implemented on binary files, leading to significant space reductions.

3.2.1 Key Size Classes Encoding

We enforce a maximum size of 1024 characters for keys. We then split the key-value pairs: First write all the values to the SSFile and record the offset for each value. Then write pairs of (key, offset of corresponding value). By setting a maximum key size, we can pad all keys smaller than 1024 to 1024. Since all offsets are also a fixed size (8 bytes), we have a fixed size for every key-offset pair. We can then perform a standard binary search on the key-offset table. If the key is found, we perform one additional seek to the offset, where we read the value. The key-offset table will be fully mapped into memory after the first iteration of the binary search search, making further iterations faster.

Padding all keys to 1024 is wasteful, so we implement a further optimization. We bucket the keys into different size classes. Each size class has its own key-offset table in the SSFile. For example, a key of size 6 would belong to the size class $[1-8]$, which means it would get padded by two characters as opposed to 1018. Thus when searching for a key, we first compute its size class, then scan to the corresponding key-offset table, where we perform the binary search. Scanning to the key-offset table is quick as the header of each table contains its length, so the number of seeks is bounded by the number of size classes.

3.2.2 Bloom Filters

Even though SSFiles are optimized for search, they still require more than one disk read [2]. In most cases, a key will not be found in an SSFile, as the database has dozens of SSFiles. We can take advantage of this by adding a bloom filter to each SSFile. A bloom filter is a memory-efficient probabilistic data structure for approximating the contents of a set. It can determine if a key does not appear in the SSFile, with a small false positive rate. Experiments showed that the bloom filter reduced the number of read operations by 48%.

The false positive rate of the bloom filter is given by

$$\epsilon = [1 - e^{-k/(m/n)}]^k$$

Where k is the number of distinct hash functions used, m is the size in bits of the filter, and n is the number of distinct elements in the filter. Since m and n are configurable, we find that $k = 3$ minimizes the function with an ϵ of 9.6%. We implement the number of hashes by performing one `SHA-256` hash, and then splitting it into k chunks.

3.3 Deletes

Similar to the Log Database, deletes pose problems with the SSTable. For example, say we add a key to the memtable, flush it to the SSFile, and then later on delete such key. SSFiles are immutable, so we cannot modify them. But the key is not present in the memtable either since we flushed it, so how do we delete it? To solve this, we keep a separate list of deleted keys, called tombstones. When we flush the memtable into the SSFile, we also flush the tombstones.

A read operation to a SSFile can then return three results: Found the key, did not find the key, or the key was deleted. If the key was not found, we continue our search on the next SSFile. If the key was deleted, we can return.

3.4 Crash Resilience

If the application crashes, the data in the SSFiles is safe as it will be already persisted on-disk. However, data that is currently in the memtable and has yet to be flushed will be lost. To prevent the loss of data we maintain a write ahead log. Similar to the Log Database, this log is append only and immutable. Every write and delete on the memtable gets recorded on the log. If the app crashes, we can walk through the log and replay all actions. When the memtable is flushed, the log is reset.

4 Benchmarks

We benchmark both databases on a Thinkpad T14S Gen 1 laptop, with 16GB of memory and a AMD Ryzen 7 PRO 4750U processor with 8 cores and 16 threads, running on Linux Kernel 6.0.6. For all of the benchmarks, unless otherwise stated, we generated a randomized workload of $20,000$ unique key-value pairs and 10 operations for each pair, where an operation can be a read, write, or delete.

4.1 Log Database vs. SSTable

Our first benchmark attempts to compare the performance of the Log Database versus the SSTable. The Log Database completed the workload in **47.32** seconds, versus **55.52** for the SSTable. It is expected that the Log Database performs faster, as simply maintaining the write ahead log for the SSTable is approximately the same amount of work as what the Log Database performs. The benefits of the SSTable come in the form of extensibility for further optimization, which we will discuss later.

[2] A constant amount of reads, to scan over the size classes, one to load the key offset table to memory, and one to read the value once the offset is found

4.2 Memtable vs. SSFile

To illustrate the differences between reads that hit the memtable versus an SSFile, we populated the memcache with a set of 4096 random key-value pairs, and then read all keys in a random order. We then flushed the memtable, and performed the same reads. The reads into the memcache completed in **109.2**ms. The SSFile reads were **40.5%** slower, completing in **153.3**ms.

4.3 Memtable BST vs. Red-Black Tree

The memtable can be implemented with any sorted data structure that allows for a sorted traversal. We compared the SSTable using a Binary Search Tree vs. a Red-Black Tree memtable. The Red-Black tree completed the workload in **53.58** seconds, slightly faster than the **55.52** seconds for the BST.

4.4 SSTable Bloom Filter

The bloom filter can improve the performance of the SSTable by preventing costly reads to SSFiles. The randomized workload with the bloom filter and the Red-Black tree completed in **49.34** seconds, **7.9%** faster than without the filter.

5 Conclusion

The extensibility of the SSTable shines when we start considering further optimizations. For example, all commercial users of SSFiles employ some kind of compaction of SSFiles. SSFiles often contain duplicate information about keys. Since we only need the most recent information, we can merge SSFiles together. This can be done via a compaction process that runs on a separate thread, analogous to a garbage collector. Furthermore, if space is a concern, we can compress SSFiles. Concurrency can also be implemented into SSTables. The immutability of the SSFiles becomes very helpful in this case, as the only data structure that needs locking is the memtable. This can be achieved via a reader-writer lock and/or a lock-free data structures.

Overall, the LogDatabase is a surprisingly performant database for its simplicity. The SSTable is a powerful and extensible database. The SSFile data structure and encoding scheme proposed in this report can be used instead of B-Trees, the classical indexing data structure used in relational databases.

References

[1] URL: https://cloud.google.com/bigtable/docs/overview.

[2] URL: https://cassandra.apache.org/_/index.html.

[3] URL: https://www.scylladb.com/glossary/sstable/.

[4] Fay Chang et al. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[5] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017.