»What I cannot create, I do not understand«

– Richard Feymann

Performance Workshop Sideproject

Implement your very own Key-Value-Store in a programming language of your choice. While you can of course code it in Bash, SQL or even Brainfuck, for best learning effect, a compiled language is recommended. The internal design of your store can be whatever you like it to be as long it fulfills the requirements below. A basic LSM (Log-Structured-Merge-Tree), as discussed in the introductionary slides, is a good and performant choice though.

Deadline: Ideally you have a rough sketch until the first workshop about CPU internals. If you did not find the time then don't worry and work in your own speed. I'm not your judge, just hoping to guide you a bit. After every workshop today you should have some ideas how to extend or improve your implementation. Try to see if you can incorparate some knowledge from the slides into your implementation:

https://sahib.github.io/misc/performance/slides/0_toc/index.html

Required minimal feature set

1. »`Get(key []byte) ([]byte, error)`« returns the value for `key` and possible errors.
2. »`Set(key, val []byte) error`« remembers `val` for `key` and returns possible errors.
3. When `Set(key, val)` returns, a subsequent `Get(key)` must immediately return the updated value.
4. The store can remember more values than there is physical memory.
5. The data must be stored persistently and loaded from disk on restart.

CPU Tasks:

1. Write benchmarks to measure performance of `Get()` & `Set()`.
2. Profile your program using a profiler and identify bottlenecks.
3. Try to fix at least one of those bottlenecks.
4. Run your benchmarks again and see if it improved.

Memory Tasks:

1. Measure the number and amount of allocation.
2. See where the allocations come from and check if you can reduce them.
3. Measure again and repeat until you identified most allocations.
4. Bonus: Implement a sparse index to lower the number of keys you can keep in memory.

I/O & Syscalls Tasks:

1. What kind of syscall does your program use? Can you identify them all?
2. Can you reduce the number of syscalls or use more efficient ones?
3. Measure the write throughput and latency of your store on full load. (`Get()` & `Set()`)
4. Make your DB crash at random points and see if all data is written (`fsync()`)

Concurrency Tasks:

1. Provide an asynchronous API for your store so users do not block.
2. Implement (segment) compression, with background I/O.
3. Try to move some I/O to the background and re-use readers.
4. Batch up writes to the database to make it more efficient.
5. Use a race detector to check your code.

Optional tasks for the motivated:

1. Implement segment compression.
2. Implement »`Delete(key) error`« using tombstones.
3. Make sure `Get()` performs well if the key does not exist.
4. Implement atomic transactions (write several values together or none at all).
5. Implement »`Snapshot(w io.Writer) error`«, which streams a consistent copy of the database to `w` (which might be stdout or a file or a socket…). This can be used as backup.
6. Implement efficient range queries ($O(\log n)$) that can list all keys with a certain prefix (i.e. Pat matches Patrick, Patricia, Pathological, …)

Info: LSM design reference

This page contains some infos about how Log-structured-Mergetree (LSM) based Key-Value-Stores are structured. The knowledge here is also part of the introductionary slides (see last part of the slide deck). This page should serve as reference therefore. **Note:** This is only one of the possible implementations of a key-value store. There are many other variations, often differing only in a few details.

Tip 1: Before you settle on a certain design you should think about what workload you want to optimize for. Is it write-heavy? Do you want to be especially fast in retrieving data? Should it be extremely resistant to crashing? Ask yourself what implications your decisions will have and experiment a bit.

Tip 2: Once you found a design you like, you should start by sketching out your code by creating your complete folder structure and create stub methods and structs. Continue by writing comments for each of them. This will help you find issues early on. Design & document your serialization format next.

Tip 3: If you would really jump right into benchmarking – which I do not recommend! – you can use my minimal, unoptimized attempt of a key-value store here: https://github.com/sahib/misc/tree/master/katta Use at your own risk.

Insertion

Every new key/value pair is inserted to an in-memory data structure. This data structure is typically a type of B-tree, but can be a hash table (if range queries are not a requirement), arrays or linked-list (if write-only workloads are desired). Once this data structure reaches a certain size, it is serialized to disk and the in-memory structure is cleared. The format should be compact, but may or may not have checksums, compression or encryption support. This data batch is called a »segment« and you will accumulate a number of segments over time.

The key-value pairs in a segment should be stored with keys ordered alphanumerically. Additionally, an **index** should be kept the in-memory that tells us what key is at what offset in the file. This index may be »sparse«, i.e. it may contain e.g. only every 10th key. The exact location of a key that is not in the index can then be interpolated using binary search. This allows the store to handle more keys than there is memory in the system at the cost of a bit more scanning and I/O.

Every key-value pair that was inserted is also appended to a Write-Ahead-Log (WAL). This log is a non-sorted segment file that is ordered by the time of insertion. This is used for crash recovery (see section below). Once a segment file was safely flushed to disk, the WAL may be truncated to retain disk space.

This concept is called »leveling«, as you first write to memory and then to disk. In theory, you might have more levels than those two if you introduce for example network-backed storage.

Querying

When a key is looked up, the in-memory data structure is first probed. If this does not yield a direct result then the index structure of each segment is probed to see if a segment contains this key and at what offset. If yes, we open the segment at the specific position and read the value from disk. Range queries are supported by checking the index for the lower and upper bound of the range and returning the key-value pairs in this offset range. Care must be taken to merge this result with the in-memory segment.

Segment Merging

There should be a background job that merges several old segments to a single one. Since those segments likely contain duplicate keys this also serves as compression scheme, improving disk usage and startup time. Since increasing number of segments also have a strong influence on query performance this should be run regularly.

Deletion

To delete keys, a special value has to be written, indicating that a key was deleted. This value is often fittingly called »tombstone«. When the querying logic encounters a tombstone as latest entry, then the key is considered as deleted. The merger will eventually remove the value. Please note: An empty value and a deleted key should be two different things!

Startup & Crash recovery

When your key-value store is loaded, the indexes of all segments need to be loaded. If the previous run of the database crashed for some reason before it was able to write the most recent segment, then the in-memory segment can be restored by loading the values from the WAL.