Parallel Log-Structure Merge Tree for Key-Value Stores

Roberto Agostino Vitillo
ra.vitillo@gmail.com

ABSTRACT

Indexing with Log-Structure Merge Trees is a core component of many modern key-value stores that need to support high rate insertions and deletions. This paper describes the implementation of a parallel LSM tree. The proposed solution achieves high insert, update and read rates and, depending on the workload, scales with the number of cores.

1. INTRODUCTION

As more data is collected than ever before by companies and research facilities, modern stores are required to deal with high insert and update ratios. The LSM tree is tailored for work-loads that experience high frequency inserts and updates over an extended period of time and only a moderate rate of retrieval operations.

A LSM tree organizes data in one or more layers tailored for the underlying storage medium. Data moves from the upper layers to the lower layers in batches using algorithms reminiscent of merge-sort. The LSM tree achieves high throughput by cascading data over time in batches from smaller but more expensive stores to larger and cheaper ones.

The proposed implementation [1] partitions the key-space into multiple independent key-value stores to scale the workload with the number of cores. Each store has its own LSM tree composed of a memory table and a hierarchical set of memory-mapped sorted string tables (SSTables).

The following sections will go over the design of the system and the performance characteristics achieved under different scenarios.

2. DESIGN

The system is designed to support multiple cores by partitioning the key-space through a hash function and distribute the operations among different workers. Each worker is pinned to a specific core to exploit cache and memory locality.

The number of partitions is configurable. Each partition represents its own independent key-value store with its own queue of tasks that is used by the client(s) to send off requests. There is one task type per operation supported:

- *GET*, used to retrieve an item from the partition;
- *ADD*, used to insert or update an entry in the partition;
- *DELETE*, used to delete an entry from the partition;
- *DESTROY*, used to delete the partition;
- *TERMINATE*, used to terminate the worker.

There is no master process and the clients push their tasks directly in the queue corresponding to the partition the key corresponds to. A future is used to retrieve the result of a *GET* operation.

Each partition is a key-value store that includes an in-memory table (Memtable) and a LSM tree represented by a hierarchical set of SSTables stored on disk.

Even though conceptually the Memtable and the SSTables can be seen together as a LSM tree, in the following paragraphs we refer to the hierarchy of SSTables stored on disk as the LSM tree.

The Memtable is implemented with a binary search tree, which allows for logarithmic insertions and retrievals. It has a configurable maximum size in bytes that determines the threshold after which it is converted to a SSTable and dumped to the first layer of the LSM tree.

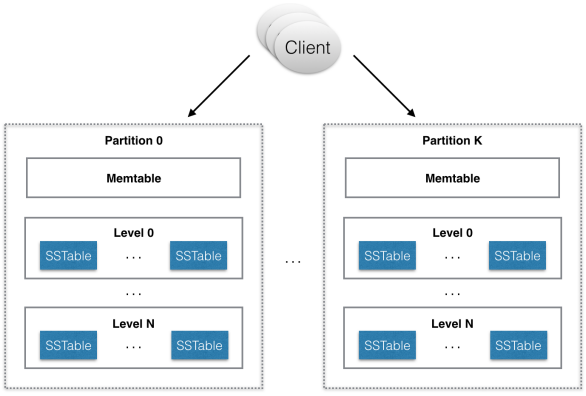

Figure 1.

Each each level contains a configurable maximum number of SSTables. When that threshold is reached, a background thread merges all the tables in the overflowing level with the overlapping ones from the next higher level. If multiple equivalent keys are encountered during merging, then only the most recent value is retained. Note that if the merge process produces more than one table then the resulting tables have contiguous keys, i.e. their key range doesn't overlap, which is a property that holds for all but the very first level.

Each level of the LSM tree has its own read-write lock to allow multiple readers or a single writer to access it. Locks are held only when the list of table pointers in a level is being updated, not when the actual merging of tables takes place, which can run without synchronization.

A SSTable is a sorted list of (key, value) pairs. Both the key and the value are stored in "buffers", i.e. arbitrary sized (up to 64KB) byte-strings prefixed with their length. A SSTable contains an array with the offsets of all the entries in the table which allows to find the position of a certain entry, assuming it exists, using binary search. The maximum size in bytes of a SSTable for a given level can be configured.

To retrieve the value for a certain key, the Memtable is queried first and if the key isn't found then each level of the LSM tree is searched in order until either a value has been found or all levels

have been queried unsuccessfully. While first level of the LSM tree requires to query all the tables within it to find the potential value associated with the key, the remaining levels need to query at most one table since their key ranges don't overlap.

Deletion is implemented by inserting a (key, value) pair in the store such that the value represents a specific deletion marker.

SSTables are memory mapped to reduce the number of copies involved when reading/writing from/to disks and leverage the OS's caching infrastructure. Though a custom tailored solution that accesses directly the underlying storage medium could be more efficient, in terms of maintenance costs and simplicity memory mapping the tables provides a clear advantage. Each level of the LSM tree can be stored in a different path of the Filesystem, which can potentially be mapped to a different storage device such as a HDD or a SSD.

3. Experimental analysis

3.1 Setup

The system was tested on a c3.8xlarge EC2 instance, with 32 virtual cores, 60 GB of memory and a 12.5 TB EBS throughput optimized HDD capable of achieving a write rate of 500 MB/s and a read rate of 250 MB/s.

Ideally the system should have been tested on bare metal as a virtualized instance can produce in some circumstances results with high variability. Furthermore, the exact specification of the hardware is not known which is going to make the analysis harder to reproduce in the future. Unfortunately, that was the only serious multi-core system available to the author.

3.2 Analysis

In the first experiment we want to show how the system scales with the number of partitions. As the most expensive operation of the system is the merging phase one would expect parallelization to shine in workloads dominated by that phase. To prove that, we used a workload with a large number of small values and a store with a small SS/Mem-table size.

In this experiment we are performing the following five steps for different number of partitions:

1. Insert 64 M random unique 10 B keys with 6 B values, for a total size of 1 GB.
2. Flush the page cache.
3. Read back all entries in the same random order of step 1.
4. Flush the page cache.
5. Update all entries in the same random order of step 1.

The LSM tree is composed of 6 levels. The SSTable size is set to 4 MB for all levels of the tree while the Memtable size was set to 8 MB. The merge threshold was set to 10^i for level i. 16 concurrent threads were used to simulate the clients. To reduce the variance each measurement shown in this section is the average of three runs.

As shown in Figure 2, adding more partitions to the store does improve the performance of the system up to a point, reaching an insert & update rate of about 600 K-items/sec and a read-rate of about 900 K-items/sec. Reads are faster than inserts in this scenario as the data on disk fits comfortably in memory. The "perf top" commands shows that this workload is dominated by the merging phase for the insert & update workloads. It's interesting to note how performance starts to saturate or even decrease when more than 8 partitions are used due to contention.

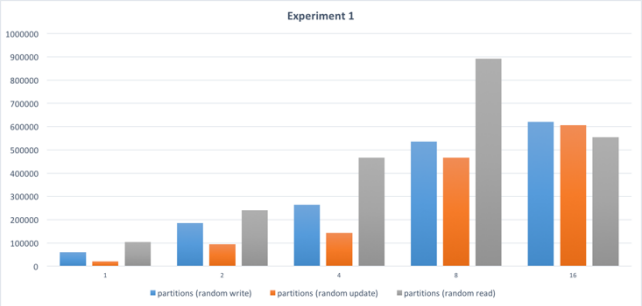

Figure 2.

In the second experiment we show that for workloads with few items and larger table sizes, adding more partitions doesn't improve the overall performance of the system.

To show that we insert, for a varying number of partitions, 1 M random unique 10 B keys with 1 KB values for a total size of about 1 GB. The LSM tree is composed of 2 levels. The SSTable size is set to 256 MB for all levels of the tree while the Memtable size was set to 1 GB. A single thread was used to feed the data to the store.

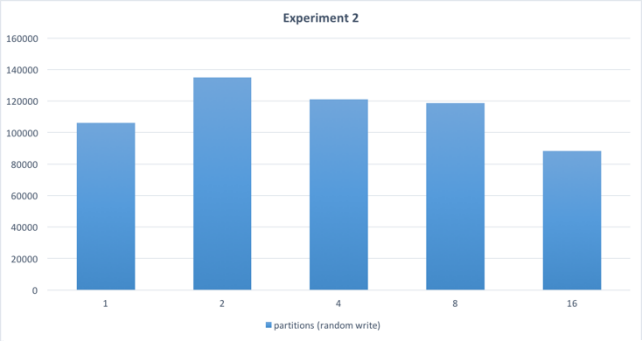

Figure 3.

As seen in Figure 3, even though adding more partitions does slightly improve the performance of the system reaching an insert rate of about 140 K-items/sec, the system doesn't scale with the number of partitions. As only few tables are involved there isn't much going on here. The data is simply copied from the Memtable to the first level of the tree and when the store is shut down the data in the first level of the tree is merged and copied to the second one. Since the write amplification is very low in this case, the store reaches a bandwidth of about 133 MB/sec which is pretty decent considering that the there are two copies involved and the theoretical maximum is 500 MB/sec. "perf top" confirms that copying data is the most expensive operation.

But what happens if more data is added to the store than there is memory? We tested this scenario in the third experiment, where we insert 100 M unique 10 B keys with 1 KB values for a total of about 100 GB. The keys were inserted both in random and sequential order in two different sub-experiments.

The LSM tree was composed of 2 levels. The SSTable size was set 1 GB for all levels of the tree while the Memtable size was set to 256 MB. A single thread was used to feed the data to the store.

Figure 4 shows that the performance for this workload is much slower than the one achieved by the previous experiment as there is a lot of swapping involved in this case. Furthermore, and unsurprisingly, there is a big difference between inserting keys randomly and in sorted order. As the latter doesn't require to read the data from the next higher level during a merge, swapping is greatly reduced resulting in a much higher insert rate.

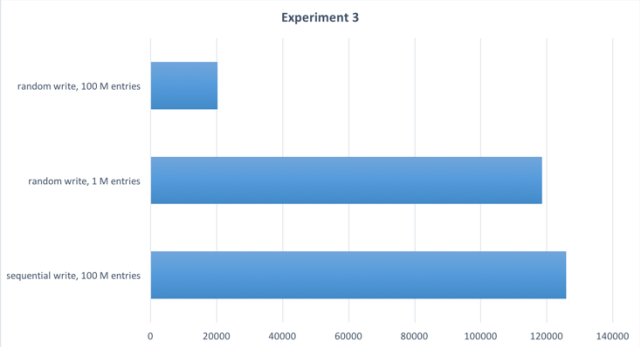

Figure 4.

More experiments should have been run given more time. For instance, the read rate should have been measured in a scenario where the dataset doesn't fit in memory. Furthermore, bigger datasets should have been used for the first two experiments and the deletion rate should also have been measured.

Conclusions

The proposed system implements a parallel LSM tree that reaches insert & update rates of 600 K-items/sec and an insert throughput of up to 133 MB/sec in the described experimental setup. The system is also shown to scale with the number of partitions depending on the workload.

4. References

[1] https://github.com/vitillo/kvstore