CSE 511 Project 2 Report

Ata Fatahi, Sushrut Shringarputale, Quinn Burke

November 19, 2018

Contents

1 Introduction

In this project we upgrade our key-value store back-end from a simple .txt file to a Log Structure Merge Tree persistent data structure. The log structure merge tree has desirable properties that make it attractive for providing indexed access to files with high insert volume. The implementation of this data structure involves two separate, but cooperating data structures: a memory-resident C0 tree, and an on-disk C1 tree. We synchronize data between the two with the use of the `c1_batch_insert()` function, which takes batches of nodes in the C0 tree and merges them with the C1 tree on disk.

This batch update occurs either when the C0 reaches its maximum threshold size, or when the timer fires a signal to dump nodes of the tree to disk. We also implement a write-ahead logging system in order to make the system more reliable against the threat of system crashes. The user can specify whether or not they want to enter recovery mode when booting the server, before listening for new client connections. Lastly, we maintain an in-memory LRU cache to exploit the locality of client requests. Exact details on our implementation are as follows.

2 Building and Running

To build the project simply run `make`. This will build the server as well as all the clients that we tested with the server.

To run the server, use

```
1 ./server 1 [r]
```

The **r** flag is optional and can be used to specify that the user wants to enter recovery mode before starting the server. To run a client, just run one of the generated client binaries, although in case the clients are not on the same system as the server, the IP address of the server may have to be manually changed in the code for that specific client.

3 Design and Implementation

3.1 C0 Design

The C0 portion of the LSM tree is implemented as an AVL tree. This tree resides in memory and provides an interface to the helper threads to interact with a portion of the database directly, without having to go to disk (except for an edge case with GET requests). Client requests may be served directly from the cache (in the case of GET requests) or directly from the C0 tree. Once the C0 tree reaches a threshold size, we initiate a dumping to the C1 tree on disk. We also schedule a timer to periodically (tunable parameter) initiate C0 dumps to disk. This section describes the implementation of the C0 tree in detail. For each of the client request types, the C0 tree operates as follows:

- **GET.** Helper threads check the LRU cache first: if found, returns the associated value with no further work, otherwise check the C0 tree. If found in the C0 tree, return the associated value. In the case of cache and C0 misses for GET requests the helper thread must go directly to disk with the `c1_get()` function and attempt to find the key.

- **PUT.** Helper threads check if the key is located in the C0 tree already: if so, update the value in place. If there is not an associated node in the C0 tree, there still might be an entry on disk for the key, so we insert a node into the C0 tree with the updated value, and when we merge on the flush of C0 to disk, we will ensure that only the most recent value associated with that key is written to the database. In the case that there wasn't an entry for the key already existing in the database, the PUT functioned simply as an INSERT. In the event of an INSERT, we also validate the size of the C0 tree to ensure it has not reached the threshold and requires dumping. This function also ensures that there will never be duplicate keys inserted into either the C0 tree or the database.

- **INSERT.** Helper threads first check if the key is located in the cache: if so, return failure immediately with a duplicate key violation. If it is not located in the cache, we check if it is located in the C0 tree, and return failure in the same case as above. If these checks pass, we complete the insert into the C0 tree, and also validate that the new size of the tree has not reached the threshold and requires dumping to disk.

- **DELETE.** Helper threads first check if the node is located in the C0 tree: if so, we update the valid flag to mark the node as invalid, so that when during the merging of C0 and C1 we are sure to *not* include this node. If the node is not located in the C0 tree, it still might exist on disk; in this case, we insert a placement node for this key into the C0 tree, with the flag set to *invalid*. During merge, this will essentially function as an update to the key stored in the database, which will be checked and ensure that the key is not written to the new version of the database file.

*Note: We also ensure that updates to C0 and C1 are reflected appropriately in the cache for all request types.

3.2 C1 Design

The C1 portion of the LSM tree is implemented as a series of immutable files containing the key-value store. While each file stays immutable, at any point in the execution, only one file is ever used as the source of truth. The cost of performing the disk accesses is amortized by the C0 tree, making the server serve requests much faster. This section describes the implementation of the C1 portion in detail.

The C1 tree contains two pieces: the database files and the search index (in code perhaps incorrectly called SSTable).

3.2.1 Database files

Each flush from C0 forces a new immutable file to be created on disk. This file contains the key-value pairs as flushed from the C0 portion. The C1 tree operates under the following assumptions:

- The flush from C0 does not contain any duplicates. In other words, any update or delete operations on keys that exist in C0 are directly handled there.

- The provided keys from C0 are in sorted order.

- The size of the key and value is under the configured limits.

Upon flushing the data to the new file, C1 triggers a merge of the last valid database file with this new file. A brand new file is created that contains the sorted, merged data from the old status of the database and the new, flushed data. Once this file is created, it is again immutable and will not ever be written to again.

3.2.2 Search Index

Naively searching through the database can be done in $O(log\ n)$ time using binary search, as the keys in the file are all sorted. The overall performance can be improved by maintaining an index that holds the offset in the file where keys starting with the same character lie. This index is only read when database is initialized. After that, the index is held in memory and persisted to disk. When a GET request is made, C1 searches through the index in memory and finds the range of offsets where the key may be found. After this, C1 will perform a binary search in that range and return the value found for the search key.

Maintaining the search index is conducted upon every flush. Once the flush is completed, C1 will look at the offsets for every key with unique starting letters and note the offsets in the index table. Then, this table is flushed to disk as well to ensure that in case the database restarts before the next flush, it knows where the offsets are.

3.3 Synchronizing C0 and C1

One of the most crucial components to this system is the `c1_batch_insert()` function. Whenever a client request is received, if it is not a GET request, then some changes will be made and needed to be flushed to disk (i.e. PUT, INSERT, or DELETE). Since we maintain an in-memory C0 tree to increase the performance of our system, these changes are the ones that will be flushed. What triggers the eventual writes to disk are either of two events: (1) the C0 tree has reached its maximum threshold, or (2) the global timer fired off a signal to dump nodes of the C0 tree.

In either case, they trigger the aforementioned function, which dumps the nodes of the C0 tree into an inorder array, and issues merges the new information with the most recent version of the C1 tree as described in Section 3.2. If the batch insert was successful, the log is flushed, as the journal entries associated with those client requests are no longer needed. Then the server can begin serving client requests again to/from the C0 tree and the LRU cache. By delaying writes using batch inserts, we are able to minimize the frequency of I/O (under our configuration limits) and are able to serve client requests much faster than the naive implementation of the database in Lab 1.

3.4 Journaling

To improve the reliability of our key-value store, we implemented a write-ahead logging system. In this implementation, we issue complete transactions to disk before any actual updates to disk will occur, so that in the event of a system crash, our portion of the key-value store that resides in-memory will not be lost. The journal entries contain the entire query string and the journal operates as follows:

1. Whenever the server receives new client requests and delegates responsibility to a helper thread, the helper thread immediately issues a transaction to disk containing information relative to the client request.

2. The `log_transaction()` function issues each field of the transaction to disk separately and the return codes are checked to ensure the write completed successfully.

3. Once the TxE is committed and successfully flushed to disk, the helper thread can continue to operate on the client request accordingly.

4. In the case of a sudden termination of the server: on server reboot, the user can specify the **r** option in order to enter recovery mode before the server begins listening to new connections. In recovery mode, the server reads the transactions (requests) that were written to disk and re-executes them in correct order as received from the client.

If at any point during execution the system crashes (especially if during a write to the log), the only part of the log file that may be malformed will be the last log entry. In this case, during recovery, the system will catch the malformed line and ignore it, which also means that the associated client request will be lost. The system will, however, be robust enough to recover and re-execute any client requests prior to the malformed entry.

*Note: A transaction remaining on disk means that the client request associated with the transaction was not yet *completely* full-filled by the server and placed into the C1 tree. Transactions remain on disk while the associated tree nodes are still in C0, and are flushed from the log once C0 is flushed to disk.

4 Evaluation

In this section, we explain the evaluation we did for our multi-threaded server with a backing log structured merge tree architecture and compare the results with those from Project 1.

4.1 Experimental Setup

Infrastructure. We first started using department lab machines, but unfortunately, the connection between two servers was so slow specially for part 2 and part 3. Since we didn't have enough time, we decided to run the experiments on a faster infrastructure to save time. Therefore, we used AWS virtual machines for our experiments. The VMs we used are two *m5d.large* that each has 2 vCPUs and 8 GB of memory and running Ubuntu Server 16.04 LTS.

Workload. The workload we use, is determined by several knobs such as the ratio of GET/PUT requests and the KEY-VALUE sizes. The GET/PUT ratio varies and is either 0.1 or 0.9. The arrival rate is following a uniform distribution with the average 50 requests per seconds with a concurrency level of 25. The Key-Value pair size also is 1KB. The popularity distribution for keys follows a Zipf distribution with 1.345675 which results in having a 90:10 popularity fraction. That is, 90 percent of the workload is targeting 10 percent of keys. We also vary the size of the C0 component (200 and 800 K/V pairs), and the size of the C1 component proptionally to C0.

Methodology. For each experiment, we have a set of 1000 unique keys which their popularity is described above. As the workloads we use consist of GET and PUT requests, we first INSERT all the keys into the database. This is two-folds, first, it fills the cache and hence brings the system into a steady state, and it prevents from getting errors and misses upon PUT and GET

requests respectively. Since 90 percent of the whole workload is targeting 10 percent of keys (100 keys), and the cache hit rate should be 90%, we set the cache size to be 100 resulting a 90% hit ratio.

4.2 Multi-Threading (MT) Evaluation

In this section, we present the results for MT architecture for 1KB sized keys, and different GET/PUT ratio including 0.1 and 0.9.

4.2.1 C0 size: 200 key-value pairs, C1 size: 200X key-value pairs

For key-value size of **1KB**, the results are as follows:

Table 1: Blocking Sockets with Multi-Threading, KV Size: 1KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	2300	5.02	5.63	5.81	6.01
0.9	9086	4.34	4.68	4.73	5.12

The request response time is as follow:

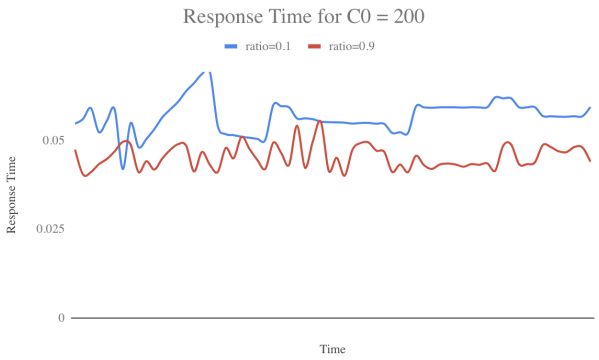

Figure 1: Response time for different GET/PUT ratios on LSMT server for 1KB key-val

Using this workload, the maximum throughput is achieved when the ratio of GET to PUT requests is set to 0.9.

Now, setting the λ to 9000 (as maximum throughput) along with a 0.9 ratio, we see the average response time is 8.64 ms.

4.2.2 C0 size: 800 key-value pairs, C1 size: 800X key-value pairs

For key-value size of **1KB**, the results are as follows:

Table 2: Blocking Sockets with Multi-Threading, KV Size: 1KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	3653	4.78	4.96	5.23	5.46
0.9	9800	4.01	4.68	4.85	5.00

The request response time is as follow:

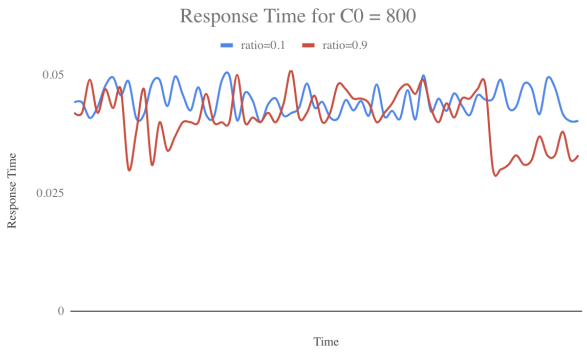

Figure 2: Response time for different GET/PUT ratios on MT server for 1KB key-val

Using this workload, the maximum throughput is achieved when the ratio of GET to PUT requests is set to 0.9.
Now, setting the λ to 9800 (as maximum throughput) along with a 0.9 ratio, we see the average response time is 9.05 ms.
Overall, using LSMT setting the size of C0 to 800 (while having 1k unique keys)nodes and the GET to PUT ratio to 0.9 we can achieve the maximum throughput.

4.3 Comparison with Project 1

The result from previous Lab are as follow:
For size 1KB:

Table 3: Blocking Sockets with Multi Threading, KV Size: 1 KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	1190	20.8	21.1	22.9	23.16
0.9	7874	3.23	3.07	5.7	7.1

Compared to Lab1, using LSMT for server with the same workload and using a large C0 (e.g. 800 nodes) can improve the throughput while the response time is comparable. Another takeaway is that when the PUT requests are more than GET requests, using LSMT with different C0 size, improves the throughput up to 3X and the response time up to 4X.

5 Conclusion

In this project we improve upon our previous implementation of the key-value store, which naively searched and appended to a text file. This required a significant amount of I/O from each helper thread, greatly decreasing the performance of our system. By using a log structured merge tree, we are able to buffer client requests in memory and serve them without necessarily having to go to disk (either in the C0 tree or from the LRU cache). In the case of cache and C0 misses for GET requests, however, we must go directly to disk.

Eventually all writes have to go to disk, and we maintain an efficient search index on disk for quick look-up in the database when necessary. When synchronizing the two components we take appropriate care in merging to ensure that no duplicate keys exist (resolved in the

C0 tree), the most recent versions of values for specific keys are the ones that actually get written to disk (resolved during merge), and that client DELETE requests are handled properly (resolved during merge). Within this methodology, we discover that having an in-memory data structure supporting our on-disk database and search index, and synchronizing data between the structures, allows us to minimize disk writes in the presence of many client requests requiring such an action. We conclude that the log structure merge tree combined with a journaling system is more effective at serving multiple clients efficiently and reliably than naive approaches that require large amount of disk I/O.