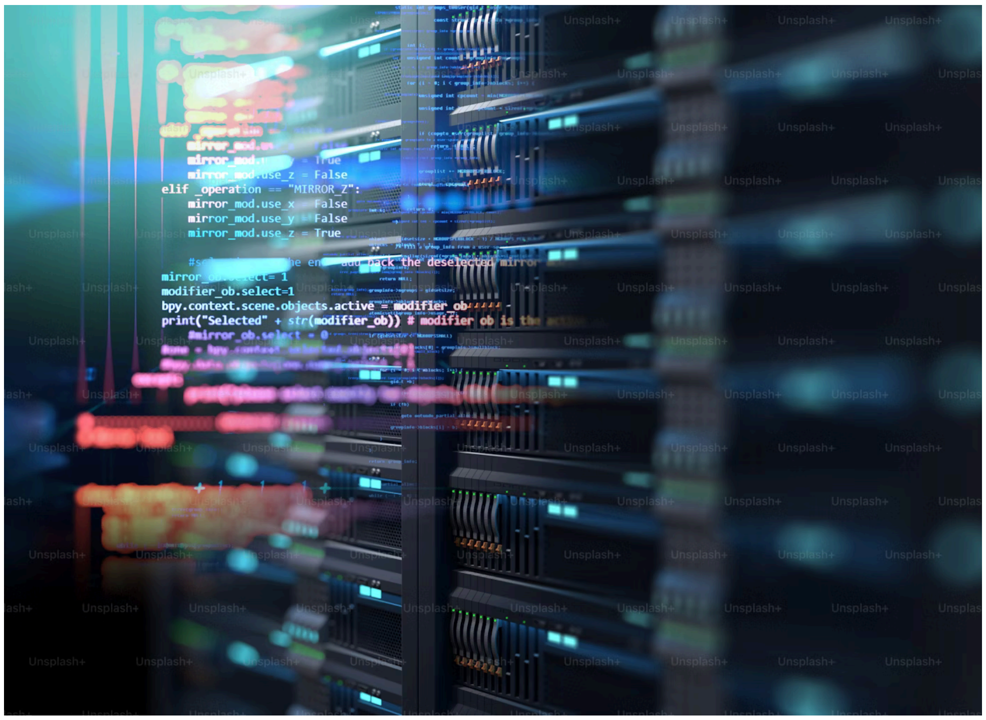
Building a Scalable Key Value Store: Implementation of LSM-Tree Architecture with In-Memory Filters



Course Instructor: Niv Dayan
Course Number: CSC443
Date: December 2, 2024

Author Detail

Andy Feng (andy.feng@mail.utoronto.ca)

Kevin You (kevin.you@mail.utoronto.ca)

Wilson Zhang (wilso.zhang@mail.utoronto.ca)

Description of Design Elements

Step 1 - Memtable and SSTs

Memtable Implementation

In the first phase of the project, we focused on implementing the memtable structure using an AVL tree, a self-balancing binary search tree. The `memtable` class, defined in `memtable.h` and `memtable.cpp`, includes methods for inserting key-value pairs, performing point queries (`Get`), and executing range queries (`Scan`). Additionally, the class provides several helper functions to facilitate tree node rotations and clearing the memtable.

The memtable constructor initializes with a size parameter that serves as a threshold. Once this threshold is reached, the memtable automatically flushes its contents into a Sorted String Table (SST), which is persisted in storage. After the flush, the `clear()` method is invoked to reset the memtable and prepare it for further data ingestion.

Key Value Store API

In this step, we also implemented the user API for the key-value store in `kvstore.h` and `kvstore.cpp`. These APIs include the required methods: `Open()`, `Close()`, `Put()`, `Get()`, and `Scan()` as specified in the project requirements. Since the `Get` and `Scan` operations must search both the memtable and the SSTs stored on disk, we also created helper functions that use binary search to retrieve data from SSTs efficiently.

The `kvstore` class also manages the process of flushing the memtable to an SST via flushMemtableToSST(). This is invoked within the `Put` method whenever the memtable reaches its size threshold, ensuring seamless transition and storage of key-value pairs.

SST and Internal Page Organization

To efficiently manage the flushing of memtable contents into storage, the process begins by creating a new empty file where the SST metadata is written first. This metadata provides essential context for interpreting the SST. Following the metadata, data is written in 4KB pages, which form the core structural unit for storage.

Each page consists of three key sections: the metadata section, which provides metadata specific to the page; the KeyOffset vector, which maps keys to offsets within the page indicating where corresponding values are stored; and the data section, which contains the actual values. To optimize space, values are stored starting from the end of the page, ensuring efficient utilization and retrieval during queries.

Step 2 - Bufferpool and Static B-trees

The core functionality of this step is pretty distinct, so we have broken them into two sections: buffer pool and static b-trees.

Buffer pool

The buffer pool was designed as a hash map. This hashmap was created individually with many of the typical hash map operations, which can be seen in our template class `HashMap.tpp`. The functions that this template supports is: insert, get, and remove. The hash map uses murmur hashing, with implementation details in `murmur3.cpp`. One thing to note: from Piazza, this file was taken from an online resource with the necessary citations found in the file. Upon encountering a collision, chaining is used. As the project handout stated, in our experiments, the buffer pool is by default instantiated with 10 MB of data. This translates to roughly 2560 pages in memory at full capacity. The hash map is instantiated with 3414 buckets, so that the load capacity is at 75% (to achieve optimal performance). The buffer pool stores the actual pages, while the hash map stores keys and pointers to the pages in memory. The eviction policy in the buffer pool is clock; clock nodes and linked lists were defined in the same file that buffer pool is (`bufferpool.cpp`). The clock eviction policy uses a circular linked list as opposed to a doubly-linked list. This was a design decision done on our end due to familiarity and simplicity. One interesting design decision that we made was implementing the buffer pool as a singleton class. This means that only one buffer pool should exist at a time, and that it is a shared object. This means that if a call is made to the buffer pool, another one should never be accidentally created. This also means that any call made to the buffer pool will be reading from the same object. The singleton design was to ensure consistency and speed.

Static B-trees

To enhance the search efficiency within SST files, we implemented a static B-tree structure. This design incorporates internal nodes in addition to the leaf nodes that represent the data pages. The nodes of the B-tree are persisted to disk by writing them in postorder into the file after all SST pages have been written. This ensures that the B-tree structure is preserved alongside the SST data. And the root node of the B-tree is specifically written at the end of the SST file. This design choice enables a straightforward retrieval of the root node's location, facilitating efficient initiation of

B-tree-based searches. To determine the appropriate search mechanism for retrieving a key—binary search or B-tree search—we included a public method in the `KVStore` class named `SetUseBTree()`. This method allows for dynamic selection of the search strategy based on performance requirements or use case specifics. Additionally, the internal nodes for the B-Tree can be stored inside the bufferpool as well to optimize IO performance.

Step 3: LSM Tree with Bloom Filters

LSM Tree Implementation

We implemented a basic LSM tree index for our key-value store with a size ratio of 2. Insertions into the LSM Tree are initiated whenever the `flushMemtableToSST()` method is called. Each time a new SST is created i.e. flushed, its filename is added to the LSM Tree index using the `addSST()` method from the LSM Tree class. If the current level in the LSM Tree becomes full, `addSST()` will trigger compaction which will recursively merge two SSTs and move the merged product into larger levels.

Supporting Updates and Deletes

With the LSM Tree in place, our KVStore API also supports out-of-place updates and deletes. Updates to a key-value pair are handled such that newer values supersede older ones during the SST compaction process. Similarly, for delete operations, a tombstone marker is inserted for the key, signaling its absence during future queries. When tombstones reach the largest level of the LSM tree, they are disposed of since older versions of the entry no longer exist.

In this phase, we enhanced the `Get` and `Scan` APIs to query the LSM Tree more effectively, traversing from smaller to larger levels. For `Get` queries, the search terminates as soon as the most recent version of a key is located, and the value is returned to the user. If the most recent version of the key is a tombstone, a negative result is returned to indicate that the key has been deleted, even if older versions exist. For `Scan` queries, the most recent version of each key within the specified range is retrieved, while tombstones are skipped to ensure that only valid entries are included in the result set.

Bloom Filter Integration

To further optimize `Get` queries, we integrated a Bloom filter for each SST, which is persisted in storage as part of the SST file. These Bloom filters work to enhance the efficiency of the `Get` workflow by pruning the search space; file accesses can be quickly bypassed if the filter query determines that the target key is not present in the SST. Whenever a call to a Bloom filter is made, we first check to see if the page containing the data is already in the buffer pool. If not, we read from storage but also read it into the buffer pool. This is so that in later calls to the Bloom filter, which is very likely depending on the work, it can be easily retrieved from memory rather than from storage.

Project Status

The project is fully functional as of the most recent commit in our repository. We made sure to test the different design elements in our development phase and they should all be integrated and work according to project specifications. No major flaws or bugs were identified in our tests so far. However, as with any software, we anticipate the possibility of minor issues emerging during extended use or in edge cases not yet encountered. The description from the previous section already went in depth so we will just provide a quick summary of the functionality for the user API calls.

API Summary:

1. **Put(Key, Value):** Successfully stores key-value pairs in the memtable and flushes to SST files when full. Out-of-place updates are fully supported.
2. **Get(Key)**: Accurately retrieves values via point queries, prioritizing the memtable and cascading to SST files when necessary.
3. **Scan(Key1, Key2)**: Performs range queries efficiently, returning the latest entries within the specified range across memtable and SSTs.
4. **Del(Key)**: Inserts tombstone markers to handle deletions, which propagate correctly through the LSM tree during compaction.
5. **Open("DB_NAME")**: Opens or initializes a database, successfully restoring prior state or creating a new one as needed.
6. **Close()**: Ensures all changes are saved and metadata is updated properly for future database use.

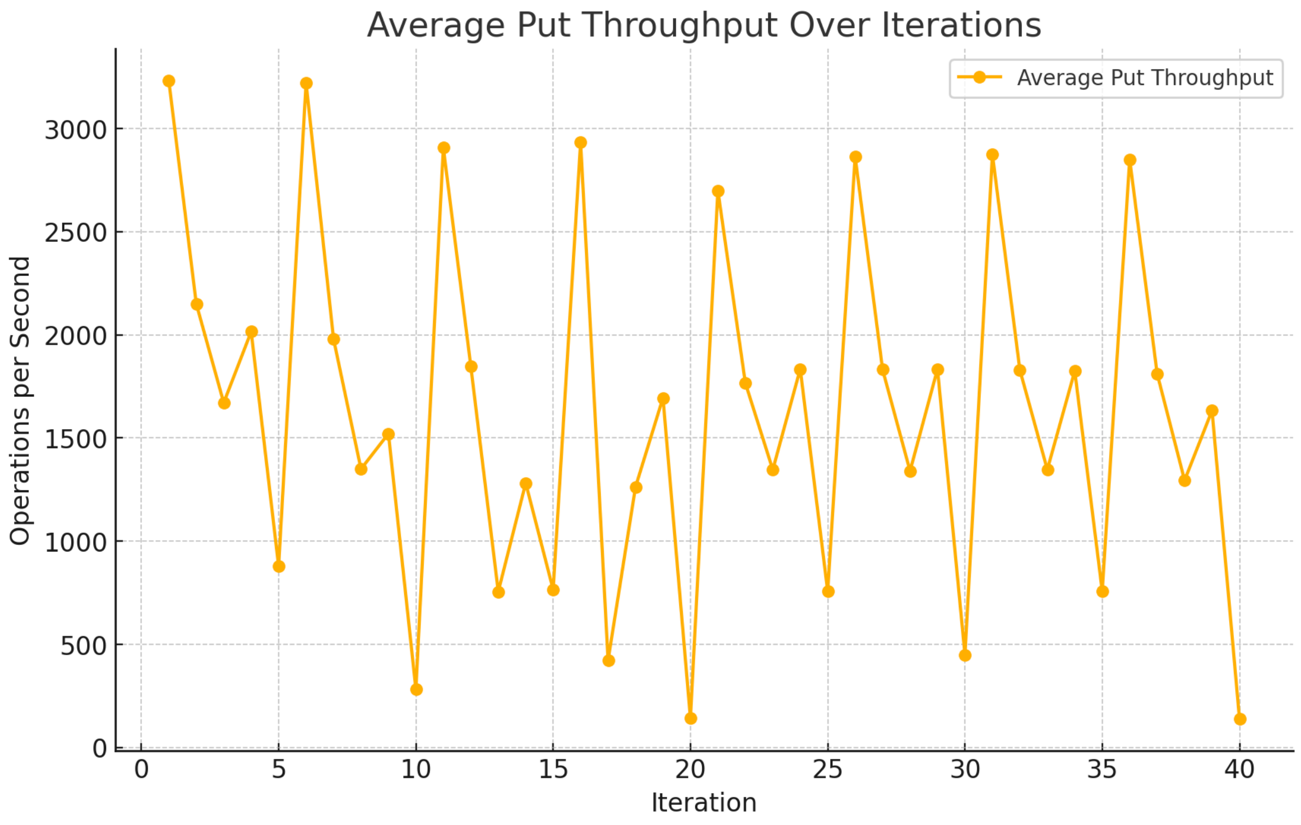
Experiments

The experiment aims to insert approximately 1GB of data. The Memtable size is configured to 1MB, the buffer pool to 10MB, and the Bloom filter is set with 8 bits per entry. Each entry (a 16-byte key-value pair) allows the Memtable to accommodate 65,536 entries.

A periodic value of 40 is defined, and during each period, we insert 1,677,722 entries with random keys in the range [1, 1,677,722 × 40]. This upper bound of 1,677,722 is calculated as $\frac{67{,}108{,}864}{40}$, where 67,108,864 is the total number of entries needed to insert 1GB of data (since 67,108,864×16 bytes = 1GB). To achieve the target data size, approximately 67,108,864 unique entries are inserted over the course of the experiment.

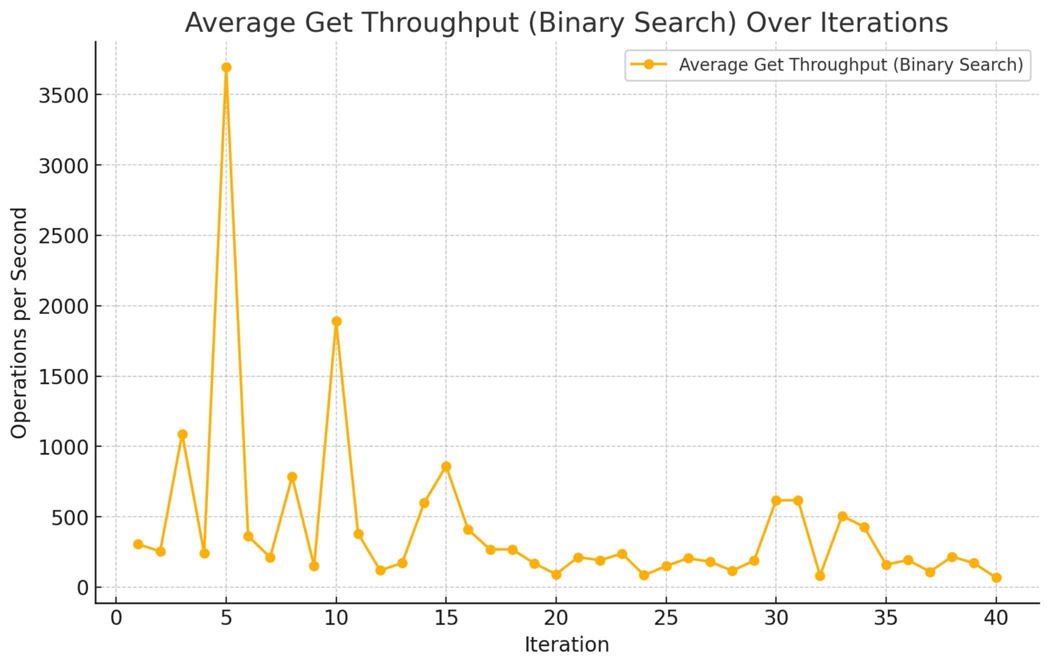
However, it is not recommended to run the experiment locally, since it takes a long time to fully execute, but our data and plots are generated by running the experiment previously. You can find the experiment script inside the experiment folder, and the experiment output is inside the build folder, named average_times.txt

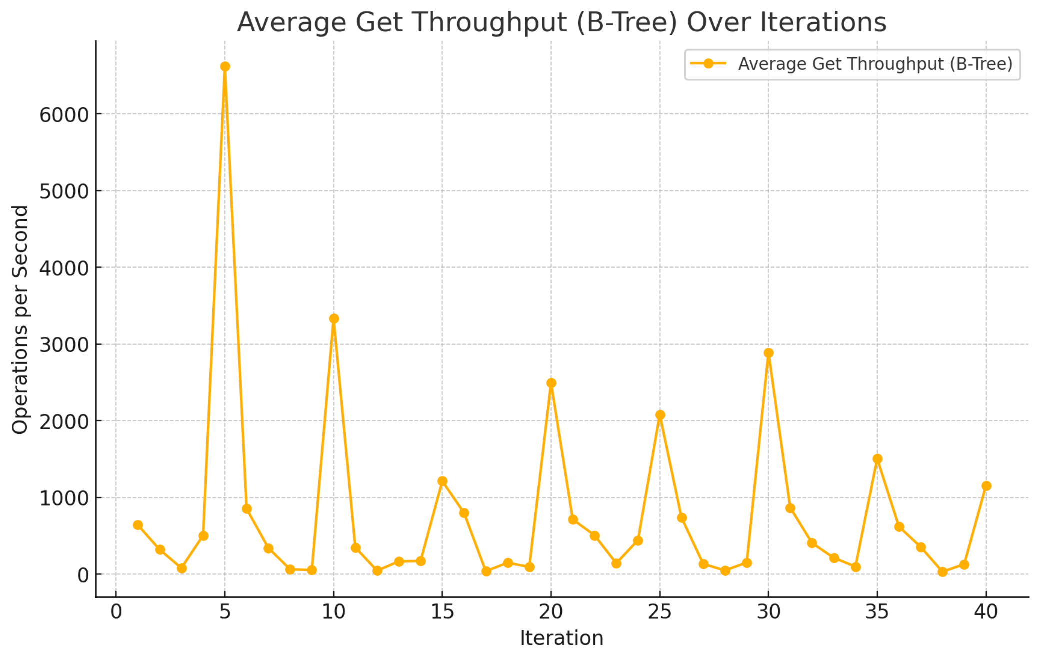
Put throughput with increasing data size



The put throughput maintains an average of around 1700 operations per second, with no noticeable downtrend in throughput. This is a good sign that our put operations do not deteriorate in performance as more data is being added. There is a noticeable pattern: after every 5 iterations, throughput spikes and then declines. This is likely due to every five iterations causing several merges to occur across different levels, which is to be expected.

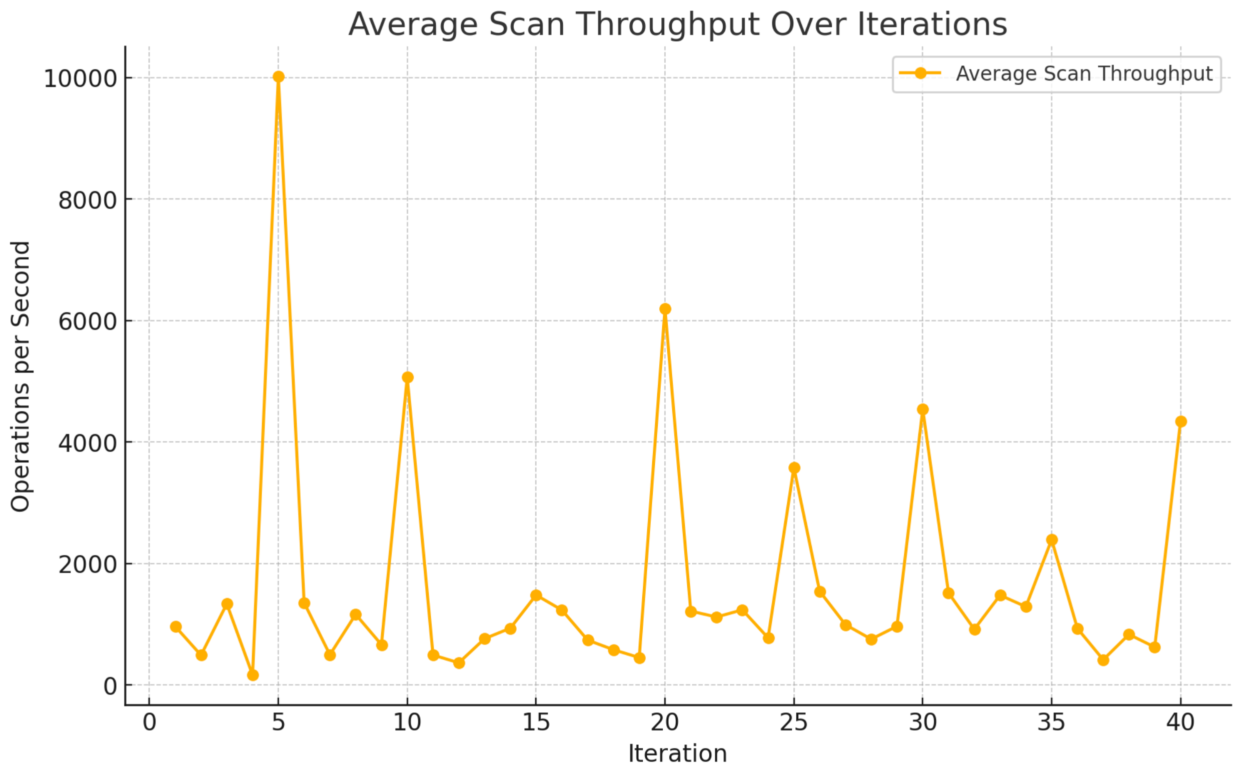
Binary search vs B-tree index with query throughput with changing data size (Get throughput)





We have GET operations using both Binary Search and with the B-tree. Notably, on average, searching for a key with B-tree is significantly higher than searching with binary search. The average for searching with binary search is around 500 operations/sec, whereas with B-tree the average hovers around 800 ops/sec. In contrast to PUTs, both GETs actually spike on the 5th iteration. This is likely due to all of the SSTs being merged into a new level, and the searching can take advantage of searching through one large SST instead of multiple SSTs across different levels.

Scan throughput with increasing data size



The SCAN operations perform similarity to the B-Tree search, but averages around 1200 operations/sec. This is likely due to scans taking advantage of sequential accesses while searches must do random accesses. This means that SCAN can take advantage of our buffer pool, and the effects are clearly notable (50% increase in throughput). The same spikes in throughput are also noticeable (on every 5th iteration). This is due to the same reasons discussed in GET.

Testing

We included unit testing for the different modules of our KVStore database, these tests can all be found under the test directory inside `unit_tests.cpp` and can be run after compiling. The tests are divided into the following categories:

1. Entity Tests: basic testing for the Page and SST class
2. AVL Tree Tests: basic testing for the memtable implemented as an AVL tree
3. Hashmap Tests: basic testing for the hashmap structure used in our project
4. Bufferpool Tests: basic testing for the buffer pool implementation
5. B-tree Tests: basic testing for the static B-tree structure
6. Blooom Filter Tests: basic testing for the bloom filter implementation
7. KVStore Tests: basic testing of user facing API inside the KVStore class

Compilation & Running

Environment Setup:

Member 1: g++ (Ubuntu 13.2.0-4ubuntu3) 13.2.0, Mac OS Sonoma 14.6.1, with VM
Member 2: g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, Windows 11, with VM
Member 3: g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, Windows 10, with WSL

Makefile uses -std=c++17

All builds must be done in the /build directory. A makefile is provided in it. To compile and create an executable, call make within this directory. Two files will be created: main and tests. Calling ../main will start a command line interface for our KV-store, and calling ../tests will run our unit tests.

The following are the available commands for the command line interface (can be viewed at any time by running the “help” command)

open <db_name> [memtable_size]	Open a database with optional memtable size
close	Close the current database
put <key> <value>	Insert or update a key-value pair
get <key>	Retrieve the value for a key
del <key>	Delete a key-value pair
scan <start_key> <end_key>	Retrieve key-value pairs in a key range
usebtree <flag>	Use Btree search or not
exit, quit	Exit the program