CSE 511: Lab 2 Documentation

Team: Game of Threads

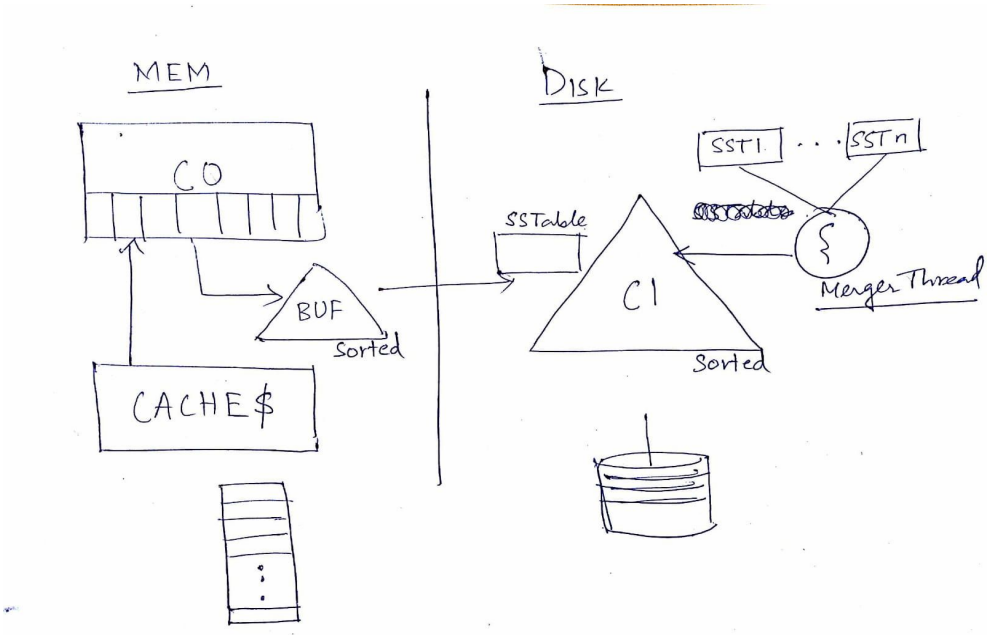
Members: Soumen Basu, Makesh Tarun Chandran, Vivek Bhasi

1. Problem Statement

In this assignment, we implemented a key-value store using Log Structured Merge Tree persistent data structure. We performed a comparative experimental performance evaluation of this implementation against the previously implemented naive key-value store. We chose to work with the Multi-threaded client server architecture.
The server supports the basic APIs (GET, PUT, INSERT & DELETE) and the same is exposed to the client as well.

2. Implementation and Design details

We have based our design on the Multi-threaded design we built for the first programming assignment. A thread is spawned for handling a client and blocking IO is used for both client and persistent storage accesses. A high level organisation of the design is presented below.



The database is organised as a LSM tree with two levels namely C0 and C1:

- C1 component is the database file in disk containing key value pairs. It is organised as a Sorted String Table based on the key
- C0 is an in-memory log containing updates to the C1 component. Once the C0 crosses a threshold size, it is merged onto the C1 component
- The merger thread is responsible for merging SSTables to C1 (This is done in O(n) time).

- Whenever C0 reached the threshold size, the memtable is flushed to an in-memory buffer and then sorted there. After sorting, the buffer is flushed to disk as SSTable.
- It is possible to have more than one SSTables that are pending to be merged with C1 but the current implementation starts the merging process as soon as one instance of C0 is available.
- While merging the deduplication and deletion of tombstoned entries happen. For Gets, we return NULL for tombstoned keys.
- We maintained a Write Ahead Log (WAL) for flushing SSTables to disks (from buffers) and merging of SSTables.
- SSTables contents are of the following form
  <key, value, last_update_timestamp, tombstone_bit> : this is handled using the LsmNode data-structure.
- The get operations may result in a binary search for the key in the file. This might incur a considerable amount of disk arm movement to reach the required key and can become the performance bottleneck when the file is big. Therefore, we also have an implementation of C1 component with an in-memory index table for the SSTable to be used when the file is large.
- Note: the methods in *utils.py* uses the storage object names as *cache* and *persistent.* All methods are generic and works for both Naive and LSM. This is just to clarify that for LSM implementation the object name *persistent* might a bit confusing. A name such as *second_level_storage* might have been more appropriate.

3. Logging and Recovery

An integral part of the LSM implementation is to ensure that recovery is possible upon an unexpected crash. We implement this aspect in our version of the LSM tree by having the transactions logged in the persistent storage just before a flush from the buffer between C0 and C1. This is done in tandem with the deduplication process that takes place during file merge. A typical entry in the log would have an indicator for transaction begin, the key-value pair after the transaction, the “end transaction indicator” and the associated timestamps with the last update time.

Note: Our design is such that the operations will be held in memory in C0 and then the buffer before being written to the log and then finally the persistent storage. The duration for which the operations reflect only in the memory is more due to the presence of the buffer (required for sorting key-value pairs). Although this choice helps us sort the key-value pairs relatively easily, we risk losing the updates in the event of a crash. However, if the crash happens after the log has been written, then recovery would be straightforward.

Sample log entry:

Txn_BEGIN
OP.MERGE([[1, '3', 1542582701.1218872, False], [4, '2', 1542582730.3951726, False], [6, '2', 1542582747.039246, False]],)
Txn_END

4. Issues faced

a. We haven't solved the backlog problem in C0 or C1.
b. Making the C0 and C1 thread-safe was a challenge. We resorted to locking. That leads to a blocking implementation. However, LSM Tree implementation wins on a write heavy workload since the writes are batched.

5. Requirements

a. Python 3
b. PickleDB for naive implementation

6. Populate DB

Can be populated using the script *populate_db.py.*

```
python populate_db.py -n <num_entries> -s <key-value-size> -d <db_name> -t <type>
```

<type>: should be "pickle" for Naive and should be "file" for LSM.

Keys are integers starting from 1. We kept keys as integers for the ease of sorting.

7. Running Servers

The naive server (db_name should end with .db):

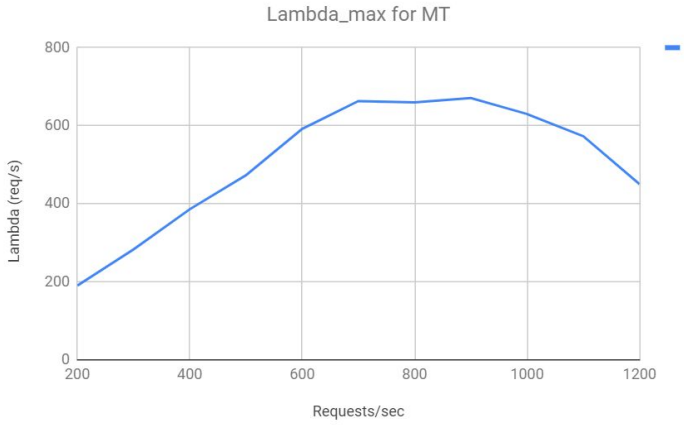
```
python multi_threaded_server.py <0.0.0.0> <port> <cache_size> <db_name>
```

For the LSM Tree based key-value server (db_name is just C1 file name):

```
python mt_lsm_server.py <0.0.0.0> <port> <cache_size> <db_name> <memtable_size>
```

8. Experimental Setup and Results

a. Server: We ran on a t2.small AWS EC2 instance (Ubuntu 18.04 w/ 20GB gp2)
b. Client Node: t2.xlarge (Ubuntu 18.04 w/ 20GB gp2)
c. Request Distribution is deterministic in nature (after 100 sec increase the number of requests by 100).
d. Skewed key popularity distribution: 10% keys make 90% requests
e. Lambda_max Measurement: We used our Lab-1 measurements for this purpose. It was measured as 663 requests/sec (for Lab-1). We approximated to lambda_max = 650 req/s



GET/ PUT Latencies for different settings:

Lambda_max = 650 req/s			1KB (Key- Value size)			
			GET Latency (ms)		PUT Latency(ms)	
			50 percentile	95 percentile	50 percentile	95 percentile
Multi Threaded Naive	GET : PUT	90:10	103	596	129	792
		10:90	127	623	134	872
Multi Threaded LSM(C0 size = 200)	GET : PUT	90:10	3.08	46.5	1.98	30.01
		10:90	2.40	59.77	1.27	24.5
Multi Threaded LSM(C0 size = 800)	GET : PUT	90:10	531	1938	110	470
		10:90	3.85	195.6	1.70	54.9

Observations:

1. Write-heavy workloads perform significantly better than read-heavy workloads for LSM Tree based implementation
2. Memtable size plays a role in determining the latency for read-heavy workloads.
3. The naive implementation is almost always outperformed by the LSM Tree implementation.