LSM TREE Implement

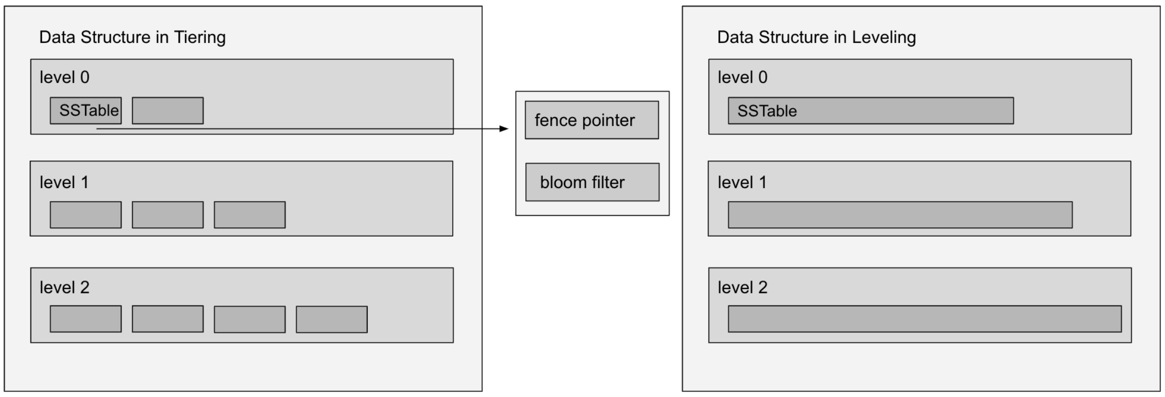
Member: Minghui Yang(U49668926), Yinan An(U59482444), Shun Yao(U0503877),

1. Expectation

A simulator of LSM Tree. Implement the fundamental operations of LSM Tree based Storage. Implement both compaction strategies — leveling and tiering.

2. Object Oriented Design & Data Structure

- MemoryTable: red black tree
- Deleted: List, sort by interval start, use binary search to optimize
- Level
    - SSTable (as Run)
    - Bloom Filter (each Table has one)
    - Fence Pointer (each Table has one)



- Other Objects:
    - Tombstone
    - Operation

3. Proposed Process

a. Init

1. Read all files & Build layered structure: Load bloom filter that stored in the SSTable metadata field into the memory, load sparse Index as fence pointer in each SSTable
2. init Memory Table
3. init WAL file

b. Write

1. Write to WAL: append-only(used for protecting from crash, the operation diary)
2. Write to the memory table : make the written data first sorted in order, ensure keys are unique in the tree. Besides a kv pair, also insert a timestamp that indicates the writing time.
3. Over threshold.
   - before flush into file, create/update bloom filter and fence pointer.
   - flush to disk as a SSTable on disk in sorted order
   - WAL is also flushed and replaced with a new one.
   - If it is over threshold for a layer, do compaction.

c. Update

The same as write, just create a new record in the database. Updating a key value is not performed in place for already existing keys. Instead, when updating a given key, just like writing a new key value pair, append a new key value pair and a timestamp to the memTable. The older value in the higher layer will not be used and they will be compacted and removed later in the phase of compaction.

d. Read

In either Tiering or Leveling, the search starts from SST in the lower level. If using Tiering Compaction, all SSTables of one level will be dealt with at first and then the next level will be searched.

Single Query

(use bloom filter and fence pointer to optimize the performance)

1. search in the memory table.
2. If don't find in memory table, search the SSTables from lower level
   - Get hash and check if it is in bitmap (bloom filter)
   - Check the fence pointer to get the SSTable that this kv pair may be in.
   - Search SSTable in the index range.
   - If find tombstone, return nullptr
3. If find, check deleted record and compare the write timestamp with timestamp in delete record, if it wasn't deleted by range delete, return, else search in the next level.

Range Query

(Only use fence pointer for to optimize the performance)

1. binary search in the memory table.
2. check the fence pointer to get all the SSTable that may contain keys in the searching range.
   - search from the every above SSTable and merge all the results.

    - If find tombstone, discard the result that may later get in higher level
    - If find the same keys with different values, discard the older updated ones(higher level).
3. iterate records in result, check deleted record and compare the write timestamp with timestamp in the delete record, if it wasn't deleted by range delete, return, else drop that record.

e. Delete

Single Delete

1. Insert a new key value pair(value to be null?) called tombstone.
2. The older value will be deleted only when there's compaction, a key value pair and a tombstone with the same keys meet, then perform delete. Not delete tombstone, it will continue compact to the higher layers.

Range Delete

Maintain a delete record vector(List) to store min/max value and timestamp of previous range. For example, a vector like [[a1, b1, t1], [a2, b2, t2]] means that at timestamp t1, we perform a range delete with keys from a1 to b1, and at timestamp t2, we perform a range delete with keys from a2 to b2. Every time we perform a delete, we add a new triple tuple to that vector. And every time we read data, we need to check the results with above vector

f. Compaction / Garbage Collection

Tigger Time: After new SST is added, determine whether compression is required.
Mechanism: (limited levels and limited size of each level)

1. Tiering: (Each layer have N SST, merge N SST per level)
    - put the new SST into the lowest level
    - If the number of SSTable(run) in a layer is over threshold, compact all SSTables in that layer and form a new SSTable(run) using merge sort
    - Insert the merged SSTable(run) into the next layer
    - Check if the number of SSTable in the next layer is over threshold, repeat from step 1 if yes.
    - update bloom filter and fence pointer after compaction: remove bloom filter and fence pointer of deleted SST, add new bloom filter and fence pointer for new SST
2. Leveling: (Each layer has 1 SST , merged 2 SST per level)
    - If the lowest layer is empty, put the new SST into the lowest level. Not empty, fetch the SSTable in the lowest level. Merge 2 SST.
    - If the size of the new SST in lowest level is over threshold. Fetch the SSTable in the next level, Merge 2 SST and put back to the level
    - If the size of the new SST in the level is over threshold. Repeat until the size is below threshold or the last level

- update bloom filter and fence pointer after compaction: remove bloom filter and fence pointer of deleted SST, add new bloom filter and fence pointer for new SST

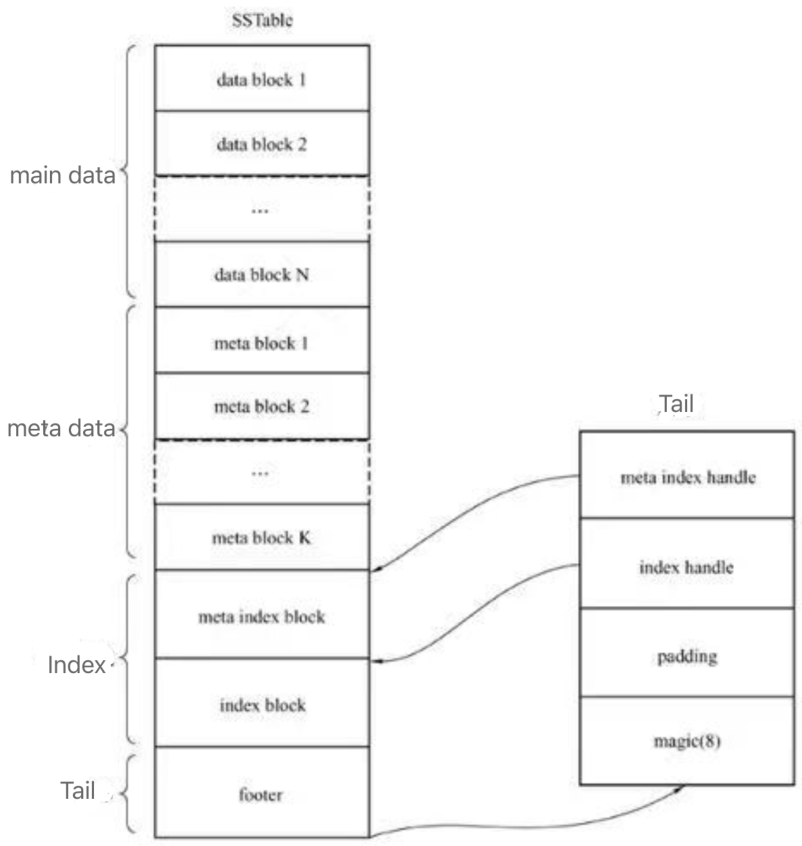
3. Data Format

DataNode in Memory Table

Node:

- key
- value
    - value
    - operation type: UPDATE, SET, GET….
    - timestamp

SSTable Format



4. Experimental Plan

Experimental data:
We can use the scripts "gen_data.py" and "gen_workload.py" to generate datasets and workloads. We can modify the data range in scripts and also define rows, dimensions when

running the script to generate data with different sizes for testing. 'gen_data.py' and 'gen_workload.py' will generate a series of operations on key values.

evaluation metrics
The simple_benchmark.cpp offers a simple benchmark that uses the executing time as a metric to evaluate the database. Besides that time, we could also use the following metrics for evaluation:
1. size of SSTable and memory table
2. bloom filter false rate
3. time of querying

5. Labor

step 0: proposal in details (collaboration)
step 1: Implement basic data structures (collaboration)
step 2: Implement operations (collaboration)