CSE291 Storage System Spr 2017
Project #2: LSM Trees, Bloom Filters, and Column-Oriented Data
Yu Huang A53207202 , Chengtao Ji A98008464

1. Introduction

In this assignment, we built a storage implementation under a NoSQL database. We adapted bloom filters to achieve look-up efficiency. Data of databases are located both in memory and on disk, using an LSM-like strategy with MemTables and SSTables, to achieve a balance between memory usage and access latency.

2. Data Model

In our implementation, each database used by one application is constructed by a group of tables. Each table consists of columns, which are the basic logical storage unit in the database.

In Memory RecentlyAccessedRowCache

We support a table-wise data structure for fast retrieval of recently accessed rows. They are stored in the limited-size priority queue so that outdated data will be deleted automatically.

In Memory MemTable

When a new table is created, corresponding MemTable objects are created per column. MemTable is responsible for storing corresponding table data in memory and flush them to disk when the size reaches certain threshold.
MemTable object keeps track of its own size. Once the size reaches the capacity set by the central config, it will flush all those <rowKey, columnValue> pairs into disk and clear its content. The rows are stored in a map for fast retrieval. Since the map has constant lookup time, we didn't implement an in-memory bloom filter to check the existence of the key.
The FLUSH operation will be covered in the API section below.

On-disk SSTable

SSTable is a per-column-based file structure on disk. In our design, SSTables are arranged by levels and have a fixed size of 16MB. In addition, starting level 1, SSTables in each level are sorted by range and their ranges don't overlap, which is called a run. The mechanism of how it is achieved will be introduced in the Implementation Section.

The format of the filename of SSTable is <table name>/<column name>/<level>_<block index>_Data.db under the <table name>/<column name>/ directory. The layout of the SSTable Data Block is as follow:

Bloom Filter: 1 MB
Row Key: string + "\n"
Column Value: string + "\n"

Time Stamp: long
Row Key
...

Every SSTable has its own bloom filter to support fast check for row keys. We don't have a level-level bloom filter because our design ensures that at most one block at each level will be accessed.

In order to fast locate which block at each level should be accessed, we also implement an Index file for every column on every level. The file is named <table name>/<column name>/<level>_Index. The format is as follow:

Block1	First Row: string + "\n"	Last row: string + "\n"
Block2	First Row: string + "\n"	Last row: string + "\n"
Block3	First Row: string + "\n"	Last row: string + "\n"

Since each block is only 16MB, it is fast enough to read the whole block into memory and do a linear search. Binary Search has worse performance because of random reads.

3. API

NoSQLInterface.java

createApplication(String applicationName)
This function will return an Application object, where user can add tables to manage as an application.

createTable(String tableName, String[] columnNames) = CREATE
Users specify the table name and column names then a table object will be created. User can then add the table to their applications. Memtables will be created per column group and the <table> directory will be created in the database directory.

Application.java

addTable(Table table)
Add a table to the current application.

getTable(String tableName)
Get the table under the application by name if exists.

Table.java

insert(Row row) = INSERT
Break the Row object into <rowKey, columnValue> pairs and perform the insertion in each column.

selectRowKey(String rowKey) = SELECT ... WHERE rowKey = ...
The table will first check the recentlyAccessedRows, if not found, it will pass the request to each column and that will be handled by MemTable then SSTables. The result from each column will be compiled to form the row.

selectRowWithColumnValue(String columnName, String columnValue) = SELECT ... WHERE column = ... (Columnwise Operation 1)
This method supports the selection of rows with one column value specified. It will go through the specified column and retrieve all qualified pairs, then go through other columns to find values with the same rowKey to construct the row, so that no need to return a whole column.

selectRowsWithColumnRange(String columnName, String operator, String target) (Columnwise Operation 2)
This method follows the similar logic as the previous one. Now it can search for rows with one column range specified.

insertColumnValues(String columnName, Map<String, String> rowKeyAndValues) (Columnwise Operation 3)
One beauty of column-based database is that we can perform column-wise operations very efficiently. This method allows insertion or updates within a column, other columns will not be affected.

deleteRowKey(String rowKey)
A new row with all column values set to null will be inserted. Based on the nature of LSM, the search for this row will return null even it is possible the data is still in the database.

update(Row row)
The updated columns with the rowKey will be inserted to corresponding MemTable. Very efficient under an LSM strategy.

JOIN
We didn't implement it. :(

4. Implementation

Bloom Filter

Bloom Filter is a data structure that can tell whether a certain key is definitely NOT present in the corresponding data. We used Bloom Filter in each data block so that the lookup can be very efficient on disk. (We didn't use it in memory as described previously.

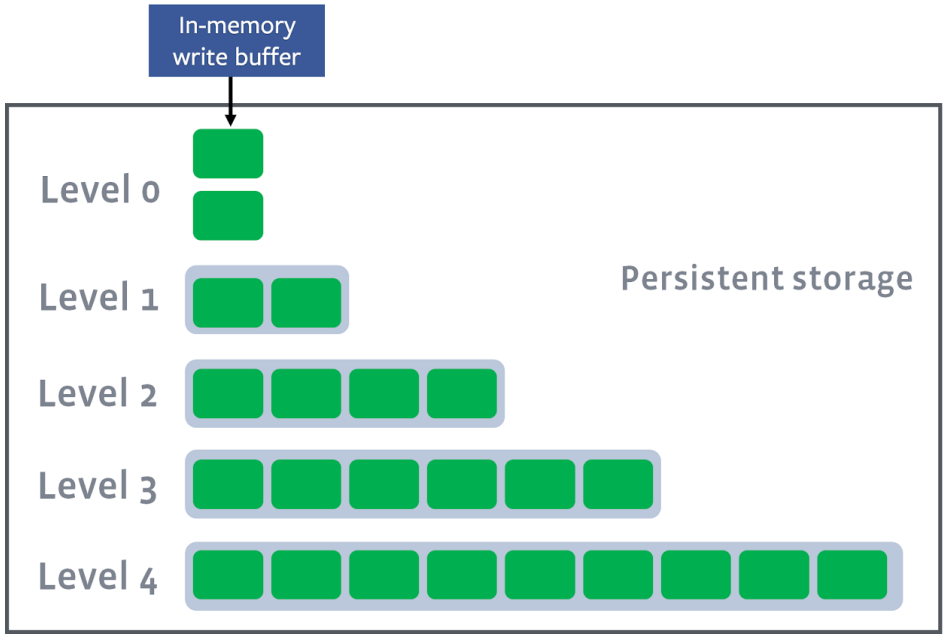
Bloom Filter is implemented by a fixed-size bit array and custom hash function to hash the key to find which bit to set. We have 1 MB bloom filter at the header of each data block, which is sufficient since it has around 8 million bits. False positive will be extremely unlikely.
When a key gets inserted, the Bloom Filter hashes the key and set that bit in the bit array. When a key gets accessed, the database will check the Bloom Filter for the hashed bit. If that bit is not set, the key is guaranteed not to be present in the data. Assuming the hash function is unbiased, if the hashed bit is set, it has a quite good chance that the key is present in the data.
The hash function is crucial to the performance of Bloom Filter. A traditional SHA1 or MD5 hash achieves good hash result but takes too long. We adapted the MurmurHash, which performs pretty well and a lot faster.

Indexing Strategy

In MemTable on memory, data are stored as Map<rowKey,columnValue>, which provides O(1) time for data access.
For SSTables on disk, there is an external index file for each level within a table:column. It has records of the start and end rowKey for each block at this level. Given the rowKey, it will be O(1) (as going through the indexBlock is extremely fast) to locate the block for searching. Our design results in sorted non-overlapping datablocks at each level, which improves the performance significantly. Without the rowKey, a global search for all SSTables across all levels is required.

Modified Leveled Compaction

The idea of Leveled Compaction is borrowed from RocksDB, where data of the same table:column are stored as SSTables of fixed size called blocks. In a Leveled Block structure, blocks are allocated into different levels starting from 0. A higher level can store 10 times more blocks.



https://github.com/facebook/rocksdb/wiki/Leveled-Compaction

Each level contains blocks in a run, which means their ranges are sorted and don't overlap. When a block needs to be pushed down, the level manager will calculate what piece of blocks in the next level

will be affected, they will be read in memory along with the pushed block then get rewritten in the same position. The blocks after that will be relocated to fix the non-overlapping sorted feature.
The rewriting process will first write to tmp files. Then the renaming will take place.
The compaction is done with the current level locked. The upper levels are still available for access (High availability), until they reach the current level, when they will be blocked.

Compression

The central config can specify two compression forms, general and Huffman(String Based). Since compressing and decompressing will significantly undermine the performance. Only lower levels (like 4 or up) will be compressed after being written or decompressed before reading.

Unit Tests & Workload Generator

We have a full suite of unit tests that ensures the correctness of every component. We integrated the workload generator with the API testing in APITest.java. The workload generator is able to generate queries for insertion, deletion, update.
Column-wise operations are benchmarked in the SSTableTest.java specifically. In APITest.java, we mainly focus on row-related operations. The insertion takes on average 9.0E-6s. RowKey selection takes 0.32s. Column selection takes 2.54s.

6. Model Evaluation & Future Development

Our implementation has a relatively complicated while highly efficient compaction algorithm, which limits the disk IO required for each query and compaction on each level. We have a lock manager to lock each level during compaction, which ensures availability on the upper levels during compaction. We very appreciate the idea of leveled compaction and do expect it to be further adapted in NoSQL databases.
One of the limitations of our project is that column queries haven't been made parallel. The advantage of column-wise design. We can also make the size of each block larger, for example to 160MB, to accommodate more data in manageable number of files. In that way, we need more graduated index structure to limit the disk IO.

7. Conclusion

In this project, we implemented a column-based storage system underlying a NoSQL database. The LSM strategy applied greatly simplify the process of deletion and modification in traditional relational database. MemTable and SSTable combination, with the help of BloomFilter and Indexing, achieves nice access performance while balancing memory usage and disk access latency.