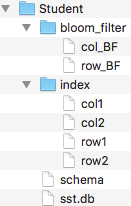
**Project2 Report**

Ziliang Li (A53209936), Qingxiang Jia (A53203082)

{zil264, q5jia}@eng.ucsd.edu

1. **Design**

**1.1 Organization of files**



Above picture describes the structure of files. Each table of a database is organized into a folder like “Student” here.

“Bloom\_filter” folder contains row-oriented BloomFilters and column-oriented BloomFilters. The *n*th line records the column names (in col\_BF) or rowkeys (in row\_BF) that appear in the *n*th SSTable. Texts in the files are separated with ‘\0’.

“Index” folder contains all index. “col1” and “row1” respectively store the column index and row index of the 1st SSTable. The first line of column index file indicates the column family name of the SSTable. The second line is the column index in the form of key-value pairs. The key is column name. The value has two parts which are respectively the start offset and the end offset of the column in SSTable. Column index are sorted by column names.

The row index are organized also in key-value pairs. The key is “rowkey.column”. The value is the start offset of the record in SSTable. All index are sorted lexicographically. Row index are sorted by “rowkey.column”.

“schema” file records the schema of the table. The first line is the name of rowkey. Each line following takes the form of “column family \0 column1 \0 column2 \0 column3 \0 …”.

“sst.db” stores all SSTables. It takes the form of “rowkey1 \0 value1 \0 rowkey2 \0 value2 ...”. SSTables are separated also with ‘\0’, because we don’t access “sst.db” directly, through index instead. Index can tell the offsets, so there is no need to records the boundaries between SSTables. The rowkey-value pairs of a column are sorted on rowkey.

**1.2 Interaction**

* **DBHandler**

A DBHander is used to handle a request to database. It gets relevant resources from Initiator according to the content of the request.

Basic logic for modification-type request is invoking the write, update or delete function of MemTable, which may further lead to dumping memtable to disk.

Basic logic for read request is reading memtable first, if missed then read cache. Then consult BlooFilters to ensure whether there are possibly newer data on disk. If so, access SSTables in the order from the most recent to the least recent. If the request is to load a cell, stop when a valid value is found on a SSTable. Values on older SSTables are void thus can be skipped. If the request is to load a row or column, it is possible to scan all SSTables since we don’t hope to miss a value saved long time ago.

**1.3 Storage**

* **Initiator**

Initiator is a manager of the system. It is instantiated when the system is started. It holds several resource pools, from which resources are assigned to the new requests to the system. Each resource in a pool is prepared for one table. Resource pools are introduced bellowing.

* **MemTablePool**

MemTable pool manages all memtables, each of which is for a table.

* **MemTable**

MemTable has two main structures to store data. One is column-oriented which provides quick column-related search. The other one is row-oriented which provides quick row-related search. A variable is used to track the current size of the data in memtable. Inserts ,updates and deletes will change the size of memtable. When it reaches a threshold, a writeToSSTable function is called. After the completion of dumping, memtable is cleared up and the current size is set to 0. MemTable supports read, write, update and delete of column-based, row-based or cell-based.

* **ReadCachePool**

ReadCache pool manages all read caches. Each read cache is assigned to a table.

* **ReadCache**

ReadCache uses FIFO scheme implemented with linked list. The basic unit of a read cache is a block. Data in a read cache are organized in hashmaps for the efficiency of search.

* **IndexPool**

Index pool stores all the index cache. Each index cache is for one table.

* **IndexCache**

An IndexCache caches the recently used index of a table in FIFO scheme. The basic unit is Index. We assume index are too huge to load into many or we hope to save for more memory used by memtable, so we don’t load the whole index to memory but load a desired part when needed and cache it.

* **Index**

An Index stores the column index and row index of a SSTable. Each time to read data from SSTables, index is used. Index structure provides function to load desired index given the id of desired SSTable.

* **BloomFilterPool**

A BloomFilterPool holds all MyBloomFilters, each of which is assigned to a table.

* **MyBloomFIlter**

A MyBloomFilter holds multiple BloomFilters, each of which corresponds to a SSTable and can be fetched via a SSTable id. The BloomFilter is taken from a open-source project. It takes about 400 codelines!

**1.4 I/O**

* **SchemaWriter**

It is used to write schema of a table to disk when a table is created.

* **SSTableWriter**

It is used to dump memtable to SSTable. It divides to-be-dumped data into blocks. It guarantees that within a block data are logically integrated, which means a rowkey-value pair (the form in which data are stored in SSTables) cannot cross blocks; the data of the different columns cannot be in a same block. Apart from writing to SSTables, it also writes bloom filters and index to disk.

* **SSTableReader**

It is used to read data from SSTables. It locates data via index. It doesn’t load a record but a block where the record lies. It calculates the start offset and end offset of a block using the offset of the record as well as the block size.

**1.5 Compaction**

The compaction is designed based on the assumption that both index and data in “sst.db” are huge. So we can’t load all data into memory then sort it and remove stale data, also we can’t load the whole index. Fortunately, we have ordered the data in SSTables and index. So we can load only a small part of data from disk at a time.

We first read the first line of each column index, from which we know which column family each SSTable belongs to. So that we can group SSTables by column family, and combine the SSTables of the same column family which finally will contract into a new SSTable. Then we basically do merge sort on SSTables. For all SSTables of a same column family, we read the index of their first column. We compare the names of columns and pick out the SSTables that have the lexicographically smallest column. Then load the first rowkey-value pairs of these qualified SSTables, compare them by rowkey and write the smallest one to a new SSTable. If there are duplicate rowkeys, it means some data are out-of-data. In this case, we compare the IDs of the SSTables that these rowkey-value pairs originate from and select the one with largest ID, i.e. the latest one; while others are discarded. Then we go to the next comparison on following rowkeys. After we are done with the merge sort of the whole column, we move to next column. Then merge sort the rowkeys of this column. Keep doing so and finish the merge sort of columns.

It is really really hard to describe the process in texts. I’d like to draw a picture but I almost exhausted time so I gave up. The key point about compaction is to do merge sort on columns; when combining columns, do merge sort on rowkeys.

1. **Test**

* **Create table**

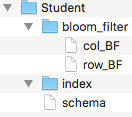
1. **Test case:**

Create table with two column families.



2) **Result:**

Relevant folders and files (some of them are empty) are created on disk.



* **Create duplicate table**

1. **Test case:**

Run the create instruction again trying to create an existing table

1. **Result:**

Refuse to create the table and print information



* **Write data and dump to disk**

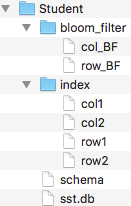
1. **Test case:**

Insert a row of record; make sure the record is greater than the threshold of memTable; threshold of memTable = 10 bytes, size of record = 26 bytes.



2) **Result:**

New SSTables are created; column index and row index are created (files “col1” and “row1” stores the column index and the row index for the first SSTable); BloomFilter files are created (col\_BF stores the columns that appear in SSTable, the *n*th line is for the *n*th SSTable; the same structure for row\_BF).



* **Read one cell from disk (miss hit on memory)**

1. **Test case:**

Restart the system (memTable and read cache are empty); read one cell from disk given a rowkey and a column. Following test is to find the name of the one whose rowkey is “001”.



2) **Result:**



* **Read one cell from disk (hit on memtable)**

1. **Test case:**

Set the threshold of the memtable as 100 bytes; insert a row of record about John; read a cell about John;

2) **Result:**



* **Read one column from disk (miss hit on memory)**

1. **Test Case:**

Insert two records at first (about John and Mary respectively); restart the system (memtable and read cache are empty); read the whole column “name”.



**2) Result:**

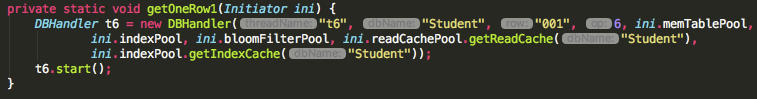
Return in form of “column : rowkey : value”.



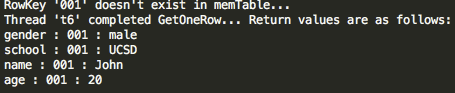
* **Read one row from disk (miss hit on memory)**

**1）Test Case:**

Read a row from disk given a rowkey.



2) **Result:**



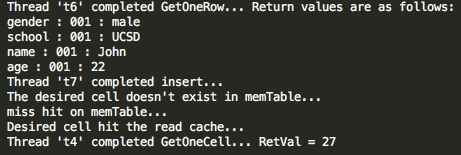
* **Hit read cache & read from disk**

1. **Test Case:**

Load a row of record about John from disk (will be cached in read cache); update the age of John from 22 to 27 and dump it to disk (by setting the threshold of memtable as 1 byte); read the age of John.

**2) Result:**

The result indicates that age 22 is cached in read cache; when read again, it hit the read cache. But the cached value 22 is stale. Age 27 is correctly read from disk.



* **Update**

1. **Test Case 1:**

Insert a row or record about John and dump it to disk (low threshold of memtable); update a cell about John (stay in memtable); read the whole record about John. Following test upgrades John’s age from 20 to 22.

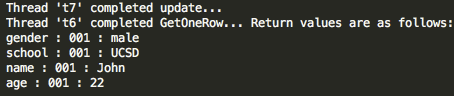
**Test Case 2:**

When doing update, also dump the update to disk rather than leave it in memtable (by setting the threshold of memtable very small such as 1 byte).



**2) Result:**

For both test cases, the age of John is read as 22, the newest one.



* **Delete**

1. **Test Case 1:**

Insert a row or record about John and dump it to disk (low threshold of memtable); delete a cell about John (stay in memtable); read the whole record about John. Following test deletes John’s age.

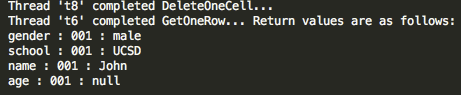
**Test Case 2:**

Dump the delete to disk; read the whole record about John.



**2) Results:**

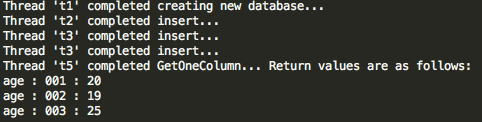
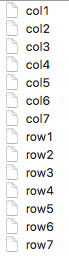
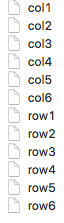
Avalue is replaced with “null” if the value is deleted. For both test cases, John’s age becomes “null” indicating being unavailable.



* **Compaction**

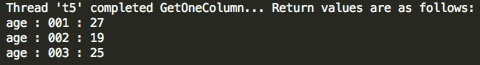
1. **Test case:**

Insert three rows of records one by one (about 6 SSTables are generated in this test case, 1st and 2nd picture); update one cell of a record and dump it to disk thus generating a new SSTable (the age of John from 20 to 27; the 3rd picture, one more SSTable is created); Apply compaction; Select the whole age column.



2) **Results:**

After compaction, only 2 SSTables remain. Each of them correspond to a column family. From the result of select, we see John’s age becomes 27. We can see the size of “sst.db” shrinks because a block holds more records thus using fewer blocks.





**3. API**

As can be seen from the Test section above, our API support equivalent functions to SELECT and WHERE. Such as:

SELECT name SELECT name SELECT \*

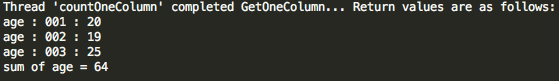
FROM Student FROM Student FROM Student

WHERE studentID = 001 WHERE studentID = 001

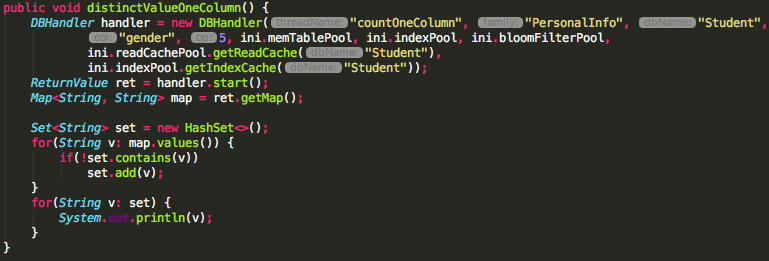
Three examples about column-oriented analysis are as follows:

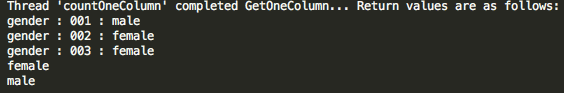
* **Sum up the values of a column**



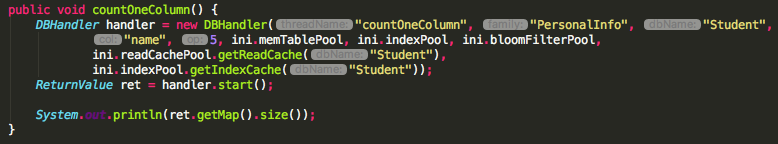


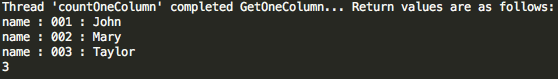
* **Distinct values of a column**





* **Count the values of a column**





**4. Discussion**

About the design, we had a misunderstanding before. we thought a database for an application only has one large table. So in our system, we mistakenly used the word “database”, it should be “table” actually because a database could have multiple tables. Each table has its own schema. Therefore, in the test cases and examples above, we created a database called “Student”, actually it should be a table called “Student”. You may create another table called “Course”. we have changed the description about “database” into “table” in this report. But we didn’t change the variable names in source code from “database” to “table” since there are too many of them. You may see some mismatch between screenshots and descriptions.

By saying that, we mean our whole system only supports one database but it supports multiple tables. For example, the whole system works as a database to store university information. It may consist of “Student” table, “Course” table, etc.

**5. Feedback**

Actually it is really a big project at least for us. We can’t imagine accomplishing it with a few hundred of codelines. We have tried our best but still don’t have time to complete the compression parts and performance evaluation. We have finished other parts and justified the correctness. Basically this is really an interesting project and we have learnt a lot from it.