Phase #3: BigTable DBMSI using Java Minibase Modules

CSE 510: Database Management System Implementation
Arizona State University
Spring 2020

by

Group #3 Team Members:
Kanishk Bashyam
Sumukh Ashwin Kamath
Baani Khurana
Rakesh Ramesh
Sreenivasan Ramesh
Ganesh Ashok Yallankar

Abstract: This report reflects the implementation of a BigTable DBMS that uses Java Minibase modules for support. The extension includes a map construct that has four fixed fields including row label, column label, timestamp, and value in replacement for tuples. B-tree based indexing methods for the map are further implemented within the disk manager package. This project implements algorithms for multiple batch inserts, individual map insert, query, row join operations, and row sort operations. To analyze the performance of different indexing techniques, query types, and buffers, the number of disk reads and writes are counted.

Keywords: Java, Minibase, Buffer Management, Disk Space Management, Heap Files, B-trees, Index, Joins, Sort, Sort-Merge Joins, Functionality, Makefile, Query, Linux, Ubuntu, Typescript, BigTable, Map, Batch Insert, Row Label, Column Label, Timestamp, Value, Stream, Read, Write, Counter, Clustering, Row Join, Row Sort, Map Insert.

TABLE OF CONTENTS

INTRODUCTION

The design of a database management system (DBMS) is crucial as it is the backbone of many applications. The storage of data can be largely complex, but by practicing different indexing techniques and strategies, simplicity can be achieved. By engaging with this project, the functionalities of a DBMS can be better understood and tradeoffs that are made can be appreciated. The value for data recovery, access patterns, query evaluation, buffer management, disk management, optimization, efficiency, minimizing cost, and maximizing performance is also learned through this experience.

With the increase in the requirement of the data storage capabilities, new methods need to be implemented. A new modern implementation is needed to be built for the new cloud environment. The big table implementation is capable of handling a large variety of different data types and allows for a software that is capable of handling applications that require a high throughput of data and scalability where the data is a non-structured form of key/value pair.

The Bigtable, in addition, offers capabilities to perform powerful back-end services. It is able to scale to the number of machines within your cluster, the administration is made easier by ensuring high durability, data replication is handled automatically when a cluster is duplicated.

Goals

The overall goals of this project include understanding and expanding BigTable DBMS functionalities to store multiple index types and implement join operators. This entails developing multiple different storage schemes for indexing and clustering. An additional goal is to allow for multiple batch insertions, different query types, individual map insertions, BigTable join called row join, and row sort operations. The last goal is to analyze the performance of the different mechanisms and gather insights to determine conclusions.

To achieve these goals, many modifications are made to the Java Minibase modules including heap package, disk manager, tuple, and iterator package, etc. By incorporating the “Map” BigTable concept as a substitute for the relational “Tuple” concept, the BigTable DBMS functionalities are successfully implemented. Additionally, performance is measured by counting the number of read and write disk accesses. A command line interface has been created to test all of the algorithms.

Terminology

Database: A database is a collection of “related” observations, a set of data, that is recorded and organized for efficient retrieval [1]. There are different types of data models that formally describe data storage mechanisms for a database such as physical models including data structures, logical models including

relational, OO, and OR, and conceptual models such as UML, ER, and Extended ER. The tradeoffs include performance and the expressive power.

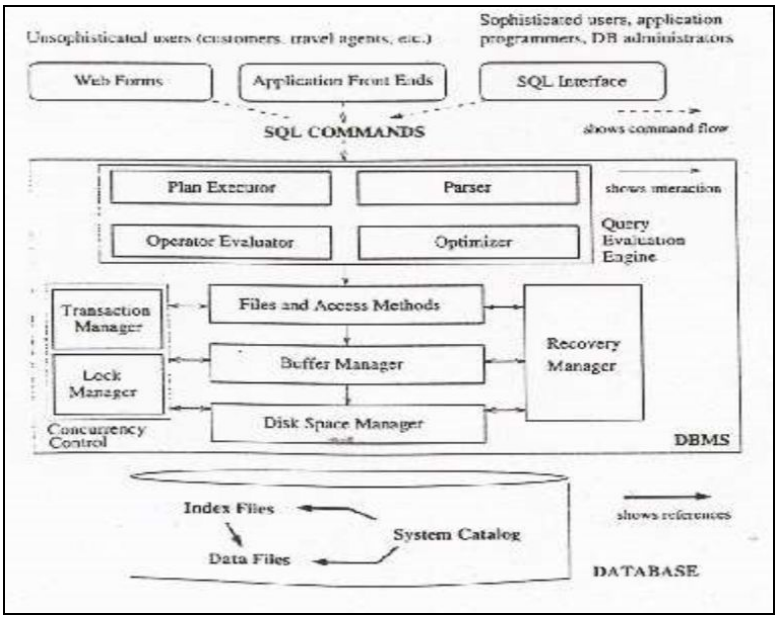


Figure 1. Architecture of a DBMS

Database Management System (DBMS): A software and hardware system for storage, retrieval, and manipulation of data [2]. The DBMS supports efficient search, queries, and automation. It also enforces constraints, enables concurrency and transactions of data, and maintains consistency and recovery. The architecture of a DBMS includes a System Catalog, Disk Manager, Buffer Manager, Recovery Manager, Concurrency Control, and Query Optimizer, etc.

Java Minibase: A relational DBMS that is traditionally row-store. It contains many functionalities described in a DBMS. These Java minibase modules serve as the foundation of a transformation into a big table DBMS using "Map" constructs instead of "Tuple" constructs.

Bigtable: Bigtable is a "distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers" [3]. It provides flexibility and high performance with sparsity, distribution, persistency, and multi-dimensionality.

Map: A construct with four fixed fields: row label (string), column label (string), timestamp (integer), and value (string). Each value in the map is an array of bytes.

$$(\text{row:string}, \text{column:string}, \text{time:int64}) \rightarrow \text{string}$$

Figure 2. Declaration of a Map Concept

Disk Space Manager: This part of DBMS focuses on storage of pages within a database such as allocation and deallocation. It conducts read and write operations to and from the disk. The Disk Space Manager (DSM) preferably needs to be robust and durable. Overall, DSM is responsible for reading from the disk,

writing to the disk, allocating pages within the database, and deallocating pages through the Database class and Page class.

Buffer Manager: The purpose of the buffer manager is to switch disk pages in and out of the main memory [1]. It writes to new pages, reads, and releases the pages to the operator. This is a critical feature and needs to be managed efficiently because there is a limited amount of room in the buffer.

File Manager: There are two main roles of a file manager: to organize the disk pages into a coherent file and to store rows into pages. The goal is to acquire and link disk pages and then organize records within the disk pages. There are two types of files: data files such as a heap file and index files.

Heap File: A type of data file where the records in the pages are unsorted [2].

Index File: A data structure that helps locate and access the data in an efficient manner [1].

B+ Tree: A tree structured index file where the tree height is balanced. The leaf pages contain data entries and are chained, while non-leaf pages have index entries [1]. B+ trees support equality and range searches. They are useful for dynamic data including insertions and deletions. An important criteria of B+ trees is to maximize fanout to allow for more utilization and pruning [2]. Additionally, it may be costly to insert large records. Therefore, bulk loading is also supported.

Indexing: A mechanism to speed up and maximize performance of queries by minimizing disk accesses within a database. It utilizes selections on the search key fields for efficient retrieval.

Query: A query is like a search to retrieve specific data from a database. There are many different types of queries. In this project, there are various parameters that determine the type of query such as the index type, attribute by which records are ordered, a row filter, a column filter, and a value filter. Each filter can either be unspecified, a single value, or a range of values.

Relational Algebra: Is a procedural query language, meant purely as an imperative definition. It Takes relations as inputs and returns a relation as an output. Relational algebra forms the basis of all database query languages. It consists of mainly five primitive operators namely, selection, projection, cartesian product, set union and finally set difference.

Relational Calculus: It is a declarative way to describe queries made up of tuple relational calculus and domain relational calculus. It should communicate what is wanted.

Join: A join operator utilizes a related column to combine rows from two or more tables together. It is like a cartesian product, but with a filter condition [2]. There are many different types of implementations for join operators including Simple Nested Loops Join, Page Oriented Nested Loops Join, Block Nested Loops Join, Index-Nested Loops Join, Sort-Merge Join, and Hash Join.

Query Evaluation: In order to evaluate a query, a query plan must be designed which is a tree of relational algebra operators that has a choice of algebra for each operator [2]. The goal is to find the best plan and avoid worse plans. Additionally, the goal is to estimate the cost of different query plans so that they can be optimized. Some algorithms to evaluate relational operators include indexing, iteration which is scanning, and partitioning which can be hashing or sorting [2].

Query Optimization: Cost can be defined as the number of disk accesses. Query optimizers use cost models to estimate query execution cost for every possible query execution plan [2]. It will utilize statistics about relations, tuples, page sizes, etc. Additionally, it will also use database information such as index and sorting. It will then choose the cheapest query plan while also remembering interesting orders which can be useful later on.

Sorting: Orders and sorting are important for query order types, duplicate elimination, sort-merge joins, and query group by [2].

Assumptions

1. We are considering the Value field as of data type string (`java.lang.String`).
2. The TimeStamp field is assumed to be of Integer data type (int) and the values in the timestamp field are considered in seconds. Millisecond values need long support and we are currently assuming only values are only in seconds (int).
3. The string length of the resultant row field from row join is assumed to be less than 25. The same length is assumed for the column field.
4. If the input string is given more than 25 characters, then the string is trimmed and only the first 25 characters are taken into consideration.
5. Since the values are stored as strings, the sorting for values will be sorting on strings and values are not considered as numbers.
6. In row join operation, The matching maps with the same latest values and columns are stored as separate maps in the resultant big table(as they differ only by timestamp).

SOLUTION

This section provides the details for the design and algorithms implemented to develop the functionalities of batch insert, map insert, query, row join, and rowsort while meeting the specifications provided.

Implementation

Changes from Phase-2

Multiple Batch Insert

In this phase of the project we allow multiple batch inserts. When records are inserted, the duplicate records are checked and if the record is already present, the record will be updated, and it will also be moved to a new type depending on the type passed. The records are also checked for the versions and the count. The record count should be less than or equal to 3 over the entire bigtable. Individual versions can be present on different types, but logically they are part of the same big table. So overall there will be at most 3 versions of the record which is stored in the bigtable.

Fixed Length Records

In the 2nd phase of the project, we had which were stored on variable length which was fixed with respect to a column. This length was obtained by taking the max length of the corresponding column. Though this was slightly efficient, this had a drawback, multiple batch inserts were not possible, if the record length was more than the column length. So in this phase of the project, we decided on keeping the record length fixed to a higher number to accommodate for fairly large strings. Although this may not be efficient and result in wasted space, this approach is simpler and we can do multiple batch inserts by keeping it fixed.

Phase-3

1: Create getCounts CLI

The getCounts command is used to return the number of maps, rows and columns stored within the database. The methods invoked on executing the command are the ***getMapCnt()***, ***getRowCnt()***, ***getColumnCnt()*** which is provided by the BigT class. The getMapCnt returns the total number of maps stored in all the 5 storage types and the method ***getRecCnt*** of the heap file is used to return the number of maps stored within each heapfile. This method originally is meant to return the number of records in file, however with the change of the data model used in our implementation, is capable of returning the number of maps. The number of distinct rows are computed by using stream to fetch all the records in the bigTable sorted by row. The number of distinct columns is also computed in a similar fashion.

The command used to get the count of maps, distinct rows and distinct columns in all the heap files is given below

```
getCounts NUMBUF
```

2: Big Table supporting multiple storage types

Our BigT implementation allows multiple storage schemes. We modified BigT and BigDB constructor and removed the type in the constructor, and whenever BigT is called, in the constructor of BigT we create 5 different heap files which support multiple storage schemes. The index files are also created along with the heap files. Different storage and index schemes are defined as follows.

- Type 1 - No index and no particular order for storing records
- Type 2 - Maps are sorted and stored according to row labels in increasing order and a btree index on row labels
- Type 3 - Maps are sorted and stored according to column labels in increasing order and a btree index on column labels.
- Type 4 - Maps are sorted and stored according to column and row in increasing order and a btree index on this combined key of column and row label.
- Type 5 - Maps are sorted and stored according to row label and value in increasing order and a btree index on a combined key of row label and value.

Records can be inserted into the table on any of the types specified above. Multiple records can be inserted in a type. Even though data is split and stored in different types, logically it is the same big table, and hence all the rules apply to the entire table and not individual heap file i.e., at any given point of time, there can be at most 3 versions of the record w.r.t a table. In order to achieve this, first the records are sorted in row column and timestamp order. The sorting is done after adding records to a temporary heap file. Once these are sorted, it is easy to find the duplicate records, and once these duplicates are found, if the count is more than 3, the oldest ones are removed and only 3 records are maintained. This pre processing step is done to increase the efficiency of the batch insert. The records, once they are pre processed, the records are processed one at a time, the records are checked for duplicates in all the existing records present in the 5 storage schemes. The records if they are present in the other schemes and if the current record needs to be inserted in the type given and the overall count is more than 3, the oldest records will be removed from the other type. Once the record is removed from the respective file, the file is kept as it is instead of sorting it. A note is made on which heapfile the record was removed. Once this process is complete for all the records. Now all the heapfiles where the records were removed are sorted again and the index is rebuilt. When multiple insertions are done on the same data for different types, the records are added to the new type and then the records in the old type is removed and then both are sorted.

The command for doing a batch insert as shown below.

```
batchinsert BTNAME1 TYPE TABLENAME NUMBUF
```

3: MapInsert

A single map can be inserted into the database. All the rules for the versions for a record apply here as well. Map insert has a similar logic as explained above. Before insertion the record is checked in all the heap files, suppose there were 3 records present and the current record's timestamp is older than the oldest timestamp of the existing records, then this record will be ignored. If the timestamp is greater than the older timestamp and the record count is 3 then the older record gets deleted and then the new record is inserted and now all the heap files are sorted and the indexes are created.

The command for map insert is as shown below.

```
mapinsert ROW COLUMN VALUE TIMESTAMP TYPE TABLENAME NUMBUF
```

4: Query

Our BigT class implementation provides a stream class which takes a row filter, column filter and a value filter. These filters can be of three types: range filter, star filter and single value match filter. This stream provides a getNext method which returns the next map matching the filter conditions. The query functionality uses this stream provided by the bigT class.

The stream is implemented as follows: We check all the 5 indexing storage types and we do the following: Depending on the indexing storage type and the filter, we set a class variable which tells if we will be scanning the entire heapfile or search using the index files. This is implemented in three phases:

Type 1: We scan the entire heap file because there are no index files for this type.
Type 2:

- If the rowFilter is *, then we scan the entire heap file. We could scan on the index file also, but the BTreeFile's new_scan method requires a lo_key and a hi_key to be passed. We currently are not storing the lowest and highest values in the metadata, and are scanning the entire file when the filter is *. The advantage of using an index file over the heap file is that the index file maintains the maps in a sorted order.
- If the rowFilter is a value or a range, we are using the BTreeFile's new_scan method to scan the index. If range, we set the lo and hi keys, else we set the low and hi to the same value.

Type 3: We perform the same as above but for the columnFilter instead of the rowFilter.
Type 4:

- If the rowFilter is * and columnFilter is *, then we scan the entire heap.
- If the rowFilter and columnFilter both are range values, We use BTreeFile's new_scan method with low_key and hi_key is based on both columnFilter and rowFilter range values delimited by "$".

- If the rowFilter is range and columnFilter is fixed value, We use BTreeFile's new_scan method with low_key based on rowFilter low value and columnFilter delimited by "$", high_key is based on rowFilter high value and columnFilter delimited by "$".
- If the rowFilter is range and columnFilter is *, we use BTreeFile's new_scan method with low_key based on rowFilter low value only, high_key is based on rowFilter high value.
- If the rowFilter is fixed value and columnFilter is range, We use BTreeFile's new_scan method with low_key based on rowFilter value and columnFilter low value delimited by "$", high_key is based on rowFilter value and columnFilter high delimited by "$".
- If the rowFilter is * and columnFilter is range, We use BTreeFile's new_scan method with low_key based on columnFilter low value, high_key is based on columnFilter high value.
- If the columnFilter is * and rowFilter is fixed, then we scan the entire heap.
- If the rowFilter is * and columnFilter is fixed value, We use BTreeFile's new_scan method with low_key and high_key both set to columnFilter.
- If the both rowFilter and columnFilter are fixed values, We use BTreeFile's new_scan method with low_key and high_key set to rowFilter and columnFilter delimited by "$".

Type 5: We perform the same above operations with rowFilter instead and columnFilter and ValueFilter instead of rowFilter respectively.

Filtering: Once we decide which file to scan, we then filter the data, and add it to a temporary heap file.

- Heapfile: If we are scanning the entire heap file and all the filters are * then we initialize the temporary heap file to the original heap file as it will have the entire data. Otherwise, we open a mapScan and iterate through the maps, and check each entry if it satisfies our filter conditions. If all the conditions are satisfied the map is added to a temporary heap file.
- Indexfile: If we are scanning the index file, we use the BTreeFile's get_next() method to get the next entry. Once we get the next entry, we check the map against the filter conditions of fields other than the field present in the Index file, and if they are all satisfied the data is added to the temporary heap file.

Sorting: Once the data is filtered, we then check the orderType to decide how to sort the data. Based on the orderType, we set the field to be sorted in *sortField.* The above filtered Heapfile or the Index File is passed to the Sort object as iterator for sorting.

Sorting is based on OrderType:

1. Sorted by sort field 1 - row field, column field and timestamp value.
2. Sorted by sort field 2 - column field, row field and timestamp value.
3. Sorted by sort field 1 - row field and timestamp value.
4. Sorted by sort field 2 - column field and timestamp value.
5. Sorted by timestamp value.

The command use

```
query TABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
```

5: BigT Join Operator

The BigT Join operator joins the two bigtables based on the row values from each of the tables respectively and stores the resulting joined maps into a new user-specified BigTable. The Inputs for the Join operator are - two BigTable names to be joined, resultant BigTable name to store the join results, column name for joining the tables and maximum limit of buffer pages to be used for join operation.

This operator involves the following methods - **storeLeftColMatch()** to filter maps matching the column name from the left table, **storeRightColMatch()** to filter maps matching the column name from the right table, **SortMergeJoin()** method is used to sort and merge the filtered results based on row value, and **StoreJoinResult()** joins the resulting rows and stores them in the resultant big table.

This operator also consists of supporting methods - **getJoinMap()** method returns the resulting map after joining the row values, **getFirstThree()** method returns the latest three maps from joined rows, and **cleanUp()** method is used to close all the bigtables and heapfiles opened for the join operation.

We have added the command-line program for the rowjoin which invokes the row join operator. The syntax for rowjoin is as follows:

```
rowjoin BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER NUMBUF
```

6: rowSort Operator

The rowSort operator takes an input table and sorts it (in nondecreasing order) based on the most recent values, for a given column. We have bigT/rowSort.java which takes in the input table, the column filter and the number of buffers. It provides a Stream object which supports a getNext method, which allows us to get a stream of rows in the required order.

To do this, we first should return all the maps which don't match the column grouped by the rows, and then the remaining rows which match the column grouped by rows and sorted by the values. We create a temporary heap file to store the intermediate results. The temporary heap files contain the data in non-decreasing order and for rows not matching the filter criteria, their values are replaced with a 0 so that they will appear at the top during the sorting process. Once the temporary heap file is created, we create a sort object on this temporary heap file to get the maps ordered by values. We finally open a new map scan object and we scan through this using the row values provided by the sort object that we created previously.

In Utils.java, we create an object of rowSort and use the getNext method to iterate through the values and store it in the output table. The command line invocation for row sort is as follows:

```
rowsort IN_TABLE OUT_TABLE COLUMN_NAME NUMBUF
```

Output and Analysis

Testing is integral to any development project and ensuring that the functionalities are working. Therefore, we have conducted tests on batch insert, map insert, query, row join, and row sort.

Batch Insert Tests

Data File 1, Type Index 1

batchinsert /home/baani/Downloads/Dataset/Data1.csv 1 test1 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  batchinsert /home/baani/Downloads/Dataset/Data1.csv 1 test1 2400
DB name =>/tmp/baani.db
Replacer: Clock

count = 9780
9779Final Records =>
==========================
IndexType 1
==========================
{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:936, Value:005916}
{RowLabel:Alabama, ColumnLabel:American_, TimeStamp:3373, Value:070590}
{RowLabel:Alabama, ColumnLabel:Anatidae, TimeStamp:2095, Value:035831}
{RowLabel:Alabama, ColumnLabel:Anatidae, TimeStamp:8508, Value:050390}
{RowLabel:Alabama, ColumnLabel:Anatidae, TimeStamp:8834, Value:014386}
{RowLabel:Alabama, ColumnLabel:Angel_sha, TimeStamp:1597, Value:089552}
{RowLabel:Alabama, ColumnLabel:Angel_sha, TimeStamp:3068, Value:012921}
{RowLabel:Alabama, ColumnLabel:Black_pan, TimeStamp:3757, Value:048182}
{RowLabel:Alabama, ColumnLabel:Bullhead_, TimeStamp:2106, Value:002635}
{RowLabel:Alabama, ColumnLabel:Carcharhi, TimeStamp:3861, Value:028070}
{RowLabel:Alabama, ColumnLabel:Chiloscyl, TimeStamp:9958, Value:097215}
{RowLabel:Alabama, ColumnLabel:Cobra, TimeStamp:3532, Value:059856}
{RowLabel:Alabama, ColumnLabel:Columbida, TimeStamp:9212, Value:011934}
{RowLabel:Alabama, ColumnLabel:Cormorant, TimeStamp:2579, Value:072264}
{RowLabel:Alabama, ColumnLabel:Cormorant, TimeStamp:3874, Value:035457}
{RowLabel:Alabama, ColumnLabel:Dolphin, TimeStamp:1765, Value:097773}
{RowLabel:Alabama, ColumnLabel:Dolphin, TimeStamp:1837, Value:059456}
{RowLabel:Alabama, ColumnLabel:Donkey, TimeStamp:8396, Value:072412}
{RowLabel:Alabama, ColumnLabel:Echinorhi, TimeStamp:5798, Value:083539}
{RowLabel:Alabama, ColumnLabel:Echinorhi, TimeStamp:6370, Value:034369}
{RowLabel:Alabama, ColumnLabel:Echinorhi, TimeStamp:6960, Value:097664}
{RowLabel:Alabama, ColumnLabel:Fish, TimeStamp:5059, Value:062393}
{RowLabel:Alabama, ColumnLabel:Fox, TimeStamp:1158, Value:071959}
```

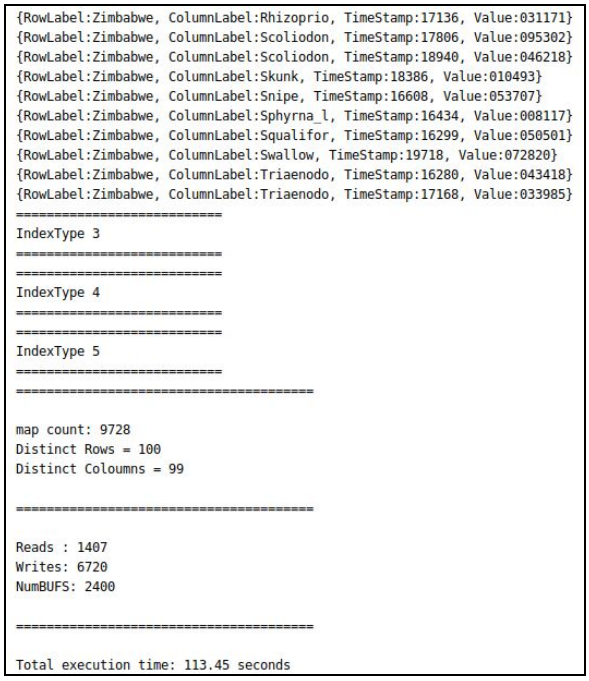
```
======================================

map count: 9777
Distinct Rows = 100
Distinct Coloumns = 99

======================================

Reads : 1139
Writes: 3864
NumBUFS: 2400

======================================

Total execution time: 91.849 seconds
```

This command inserts data file 1 into a new table with a type index of 1 which means there is no index. The results support and verify that this index type is correctly working as shown in the output above. The number of buffers is 2400 which shows a lower number of reads and writes and fast execution time.

Data File 2, Index Type 2

batchinsert /home/baani/Downloads/Dataset/Data2.csv 2 test2copy 2400

```
miniTable>  batchinsert /home/baani/Downloads/Dataset/Data2.csv 2 test2copy 2400
DB name =>/tmp/baani.db
Replacer: Clock
```

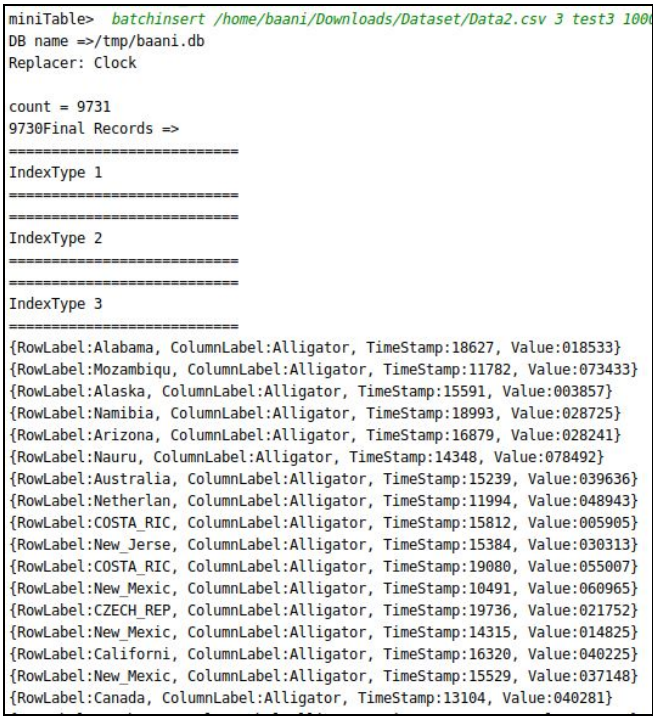


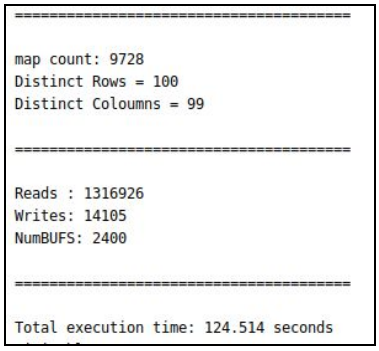
This command inserts data file 2 into a new table with a type index of 2 which indexes by row label and the maps are sorted by row label in increasing order. The results support and verify that this index type is correctly working as shown in the output above with “Zimbabwe” row label records altogether . The number of buffers is 2400 which shows a lower number of reads (1487 count) and writes (6720) and relatively fast execution time for both indexing and sorting.

Data File 2, Index Type 3

Change Num Bufs to 1000:

batchinsert /home/baani/Downloads/Dataset/Data2.csv 3 test3 1000





This command inserts data file 2 into a new table with a type index of 3 which indexes by column label and the maps are sorted by column label in increasing order. The results support and verify that this index type is correctly working as shown in the output above with “Alligator” column label records altogether. The number of buffers was also tested and lowered to 1000 from the standard 2400. The output shows a large increase, more than 50%, in the number of reads (1316926 count) and writes (14105 count). The execution time also takes longer time compared to 2400 buffers.

Change Num Bufs to 400:

batchinsert /home/baani/Downloads/Dataset/Data2.csv 3 test3copy 400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  batchinsert /home/baani/Downloads/Dataset/Data2.csv 3 test3copy 400
DB name =>/tmp/baani.db
Replacer: Clock

count = 9731
9730Final Records =>
============================
IndexType 1
============================
============================
IndexType 2
============================
============================
IndexType 3
============================
{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:18627, Value:018533}
{RowLabel:Mozambiqu, ColumnLabel:Alligator, TimeStamp:11782, Value:073433}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:15591, Value:003857}
{RowLabel:Namibia, ColumnLabel:Alligator, TimeStamp:18993, Value:028725}
{RowLabel:Arizona, ColumnLabel:Alligator, TimeStamp:16879, Value:028241}
{RowLabel:Nauru, ColumnLabel:Alligator, TimeStamp:14348, Value:078492}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:15239, Value:039636}
```

```
====================================

map count: 9728
Distinct Rows = 100
Distinct Coloumns = 99

====================================

Reads : 6842285
Writes: 25380
NumBUFS: 2400

====================================

Total execution time: 145.51 seconds
```

This command performs the same function as the previous test, but solely is different because it tests a smaller number of buffers and its effect on performance. The number of buffers has decreased even more than before from 1000 to 400. The result shows a significant increase in the number of reads (6842285 count) and writes (25380 count). The execution time also takes longer time with 145.21 seconds compared to previous which was around 124 seconds.

Data File 3, Index Type 4

batchinsert /home/baani/Downloads/Dataset/Data3.csv 4 test4 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  batchinsert /home/baani/Downloads/Dataset/Data3.csv 4 test4 2400
DB name =>/tmp/baani.db
Replacer: Clock

count = 9733
9732Final Records =>
============================
IndexType 1
============================
============================
IndexType 2
============================
============================
IndexType 3
============================
============================
IndexType 4
============================
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:23489, Value:042328}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:24155, Value:074844}
{RowLabel:Arizona, ColumnLabel:Alligator, TimeStamp:20673, Value:045197}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:24979, Value:044182}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:27181, Value:058678}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:28443, Value:056761}
{RowLabel:Barbados, ColumnLabel:Alligator, TimeStamp:22172, Value:051484}
```

```
====================================

map count: 9732
Distinct Rows = 100
Distinct Coloumns = 99

====================================

Reads : 1491
Writes: 7221
NumBUFS: 2400

====================================

Total execution time: 141.961 seconds
```

This command inserts data file 3 into a new table with a type index of 4 which is indexed by a combined key of column label and row label. The maps are sorted by the combined key in increasing order. The results support and verify that this index type is correctly working as shown in the output above with “Alligator” column label records organized by alphabetically sorted row labels such as “Alaska,” “Arizona,” “Australia,” “Barbados,” etc. The number of buffers was returned to the standard 2400 which provides the best performance. The output shows a very small number of reads (1491 count) and writes (7221 count). The execution time is also pretty fast, considering this is a more complex indexing scheme which goes to show that the number of buffers plays a significant factor.

Data File 3, Index Type 5

batchinsert /home/baani/Downloads/Dataset/Data3.csv 5 test5 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  batchinsert /home/baani/Downloads/Dataset/Data3.csv 5 test5 2400
DB name =>/tmp/baani.db
Replacer: Clock

count = 9733
9732Final Records =>
==========================
IndexType 1
==========================
==========================
IndexType 2
==========================
==========================
IndexType 3
==========================
==========================
IndexType 4
==========================
==========================
IndexType 5
==========================
{RowLabel:Alabama, ColumnLabel:Fish, TimeStamp:24064, Value:001090}
{RowLabel:Alabama, ColumnLabel:Hyena, TimeStamp:20513, Value:001872}
{RowLabel:Alabama, ColumnLabel:Gerbil, TimeStamp:28695, Value:002456}
{RowLabel:Alabama, ColumnLabel:Giraffe, TimeStamp:20314, Value:002599}
{RowLabel:Alabama, ColumnLabel:Carcharhi, TimeStamp:28913, Value:002656}
{RowLabel:Alabama, ColumnLabel:Zebra, TimeStamp:20146, Value:003420}
{RowLabel:Alabama, ColumnLabel:Swallow, TimeStamp:28839, Value:009499}
{RowLabel:Alabama, ColumnLabel:Moose, TimeStamp:28233, Value:010846}
{RowLabel:Alabama, ColumnLabel:Badger, TimeStamp:24406, Value:012823}
{RowLabel:Alabama, ColumnLabel:Donkey, TimeStamp:21050, Value:013754}
{RowLabel:Alabama, ColumnLabel:Hemitriak, TimeStamp:26574, Value:015247}
{RowLabel:Alabama, ColumnLabel:Parmaturu, TimeStamp:28853, Value:016426}
{RowLabel:Alabama, ColumnLabel:Echinorhi, TimeStamp:21649, Value:017370}
```

```
{RowLabel:Zimbabwe, ColumnLabel:Lamna, T
========================================

map count: 9732
Distinct Rows = 100
Distinct Coloumns = 99

========================================

Reads : 1498
Writes: 7116
NumBUFS: 2400

========================================

Total execution time: 126.791 seconds
```

This command inserts data file 3 into a new table with a type index of 5 which is indexed by a combined key of row label and value. The maps are sorted by the combined key in increasing order. The results support and verify that this index type is correctly working as shown in the output above with “Alabama” row label records sorted alphabetically and by increasing sorted value labels such as “001090,” “001872,” “002456,” “002599,” etc. The output shows a very small number of reads (1498 count) and writes (7116 count). The execution time is fast, remembering this is a more complex indexing scheme with combined key which again demonstrates how an optimal number of buffers can increase performance.

Multiple Batch Insert with Different Index Types

batchinsert /home/baani/Downloads/Dataset/Data3.csv 5 test3 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  batchinsert /home/baani/Downloads/Dataset/Data3.csv 5 test3 2400
DB name =>/tmp/baani.db
Replacer: Clock
```

```
{RowLabel:Zimbabwe, ColumnLabel:Fox, TimeStamp:20616, Value:089289}
{RowLabel:Zimbabwe, ColumnLabel:Black_pan, TimeStamp:25698, Value:089361}
{RowLabel:Zimbabwe, ColumnLabel:Crow, TimeStamp:24473, Value:089368}
{RowLabel:Zimbabwe, ColumnLabel:Peafowl, TimeStamp:25841, Value:089590}
{RowLabel:Zimbabwe, ColumnLabel:Lamna, TimeStamp:27946, Value:090030}
{RowLabel:Zimbabwe, ColumnLabel:Otter, TimeStamp:29061, Value:092800}
{RowLabel:Zimbabwe, ColumnLabel:Wombat, TimeStamp:25816, Value:093690}
{RowLabel:Zimbabwe, ColumnLabel:American_, TimeStamp:24518, Value:093881}
{RowLabel:Zimbabwe, ColumnLabel:Reindeer, TimeStamp:23259, Value:093953}
{RowLabel:Zimbabwe, ColumnLabel:Rhizoprio, TimeStamp:24582, Value:094425}
{RowLabel:Zimbabwe, ColumnLabel:Peafowl, TimeStamp:29181, Value:094644}
{RowLabel:Zimbabwe, ColumnLabel:Hamster, TimeStamp:20993, Value:095522}
{RowLabel:Zimbabwe, ColumnLabel:Cheetah, TimeStamp:27066, Value:096002}
{RowLabel:Zimbabwe, ColumnLabel:Goldfinch, TimeStamp:26206, Value:096270}
{RowLabel:Zimbabwe, ColumnLabel:Pristioph, TimeStamp:29502, Value:096715}
{RowLabel:Zimbabwe, ColumnLabel:Lamniform, TimeStamp:28878, Value:096781}
{RowLabel:Zimbabwe, ColumnLabel:Human, TimeStamp:26028, Value:097403}
{RowLabel:Zimbabwe, ColumnLabel:Heron, TimeStamp:24028, Value:097777}
{RowLabel:Zimbabwe, ColumnLabel:Goose, TimeStamp:23984, Value:099348}
{RowLabel:Zimbabwe, ColumnLabel:Lamna, TimeStamp:22011, Value:099844}
=====================================

map count: 17642
Distinct Rows = 100
Distinct Coloumns = 99

=====================================

Reads : 15509
Writes: 16498
NumBUFS: 2400

=====================================

Total execution time: 386.187 seconds
```

Here *test3* table has the second data file with index type 3. These results demonstrate multiple batch insert with index type 5 where the third data file is being inserted into a table that already has a data file. The results also show how multiple index types can successfully be stored within the same table. The output above verifies the correctness of this multiple batch insert functionality by showing the total map count of the table which is 17642 and the extension of all the different “timestamp” and “value” fields.

Map Insert Tests

Insert into Existing Table

mapinsert Idaho Dog 2222222 60000 5 test2copy 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  mapinsert Idaho Dog 2222222  60000 5 test2copy 2400
Replacer: Clock

{RowLabel:Idaho, ColumnLabel:Dog, TimeStamp:60000, Value:2222222}
Inserted Successfully.
Total execution time: 0.417 seconds
```

This command tests the functionality of inserting one new map into an existing table. This also shows that multiple indexing types can be stored within one table. The above output prompts a message that the insertion was successful. This is validated in a test within the query section. The total execution time was 0.417 seconds which is fast for retrieving an existing table and finding the correct slot to insert.

Insert into New Table

mapinsert Iowa Cat 40 1000 4 test6 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  mapinsert Iowa Cat 40 1000 5 test6 2400
Replacer: Clock

{RowLabel:Iowa, ColumnLabel:Cat, TimeStamp:1000, Value:40}
Inserted Successfully.
Total execution time: 0.197 seconds
```

This command tests the functionality of inserting one new map into a new table. This demonstrates that the creation of a new table is also being performed within this method. The above output prompts a message that the insertion was successful. This is further validated in a test within the query section. The execution time was 0.197 seconds which is fast.

Query Tests

Verify Results from Map Insert into Existing Table

Returns Unspecified Records with Order Type 2:

query test2copy 2 * * * 2400

```
miniTable>  query test2copy 2 * * * 2400
Replacer: Clock

{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:18627, Value:018533}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:15591, Value:003857}
{RowLabel:Arizona, ColumnLabel:Alligator, TimeStamp:16879, Value:028241}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:15239, Value:039636}
```

```
{RowLabel:Ukraine, ColumnLabel:Crow, TimeStamp:18635, Value:020325}
{RowLabel:Uzbekista, ColumnLabel:Crow, TimeStamp:17781, Value:024999}
{RowLabel:Virginia, ColumnLabel:Crow, TimeStamp:19210, Value:037081}
{RowLabel:WESTERN_S, ColumnLabel:Crow, TimeStamp:16563, Value:021734}
{RowLabel:WESTERN_S, ColumnLabel:Crow, TimeStamp:18204, Value:089204}
{RowLabel:Washingto, ColumnLabel:Crow, TimeStamp:12034, Value:078414}
{RowLabel:Zambia, ColumnLabel:Crow, TimeStamp:18107, Value:000983}
{RowLabel:Idaho, ColumnLabel:Dog, TimeStamp:60000, Value:2222222}
{RowLabel:Alaska, ColumnLabel:Dogfish, TimeStamp:11146, Value:036589}
{RowLabel:Barbados, ColumnLabel:Dogfish, TimeStamp:16703, Value:009851}
{RowLabel:Barbados, ColumnLabel:Dogfish, TimeStamp:16752, Value:062144}
```

This test validates that the *mapinsert* function into an existing table was correctly implemented by finding the insertion within the table. The new animal was added: “Dog”. Additionally, with order type 2 specified, the results are first ordered in column label, row label, and then timestamp. The output above demonstrates that this was successful with column label “Crow”, “Dog”, “Dogfish” alphabetically sorted. Next, within multiple “Crow”, the row label is sorted next shown by “Virginia”, “WESTERN_S”, etc. Lastly, timestamp is sorted as shown within “WESTERN_S” with timestamps of “16563” first and then “18204”. Overall, the results show that both map insert into an existing table and order type 2 are working correctly.

Verify Results from Map Insert into New Table

Returns Specific Record

query test6 2 Iowa Cat 40 2400

```
miniTable>  query test6 2 Iowa Cat 40 2400
Replacer: Clock

{RowLabel:Iowa, ColumnLabel:Cat, TimeStamp:1000, Value:40}

=====================================

Matched Records: 1
Reads : 27
Writes: 0

=====================================

Total execution time: 0.02 seconds
```

This test validates that the *mapinsert* function into a new table was correctly implemented by finding that the new table exists and the insertion is able to be retrieved. The new animal, “Cat”, was successfully added as shown above. Overall, the results show that map insert into a new table is correctly working.

Verify Multiple Batch Insert

These following tests are validated on the table *test3* that has two data files through multiple batch inserts. Each query tests different combinations of filters and order types.

Return Records with Range Row Filter and Order Type 1:

query test3 1 [Virginia,Wisconsin] * * 2400

```
miniTable>  query test3 1 [Virginia,Wisconsin] * * 2400
Replacer: Clock

{RowLabel:Virginia, ColumnLabel:Alligator, TimeStamp:25920, Value:069117}
{RowLabel:Virginia, ColumnLabel:American_, TimeStamp:18002, Value:004438}
{RowLabel:Virginia, ColumnLabel:American_, TimeStamp:26039, Value:026327}
{RowLabel:Virginia, ColumnLabel:Angel_sha, TimeStamp:18905, Value:039193}
{RowLabel:Virginia, ColumnLabel:Apristuru, TimeStamp:16349, Value:057545}
{RowLabel:Virginia, ColumnLabel:Apristuru, TimeStamp:29735, Value:065350}
{RowLabel:Virginia, ColumnLabel:Baboon, TimeStamp:21866, Value:072239}
{RowLabel:Virginia, ColumnLabel:Badger, TimeStamp:12025, Value:029991}
{RowLabel:Virginia, ColumnLabel:Beaver, TimeStamp:18404, Value:025067}
{RowLabel:Virginia, ColumnLabel:Beaver, TimeStamp:20422, Value:037374}
{RowLabel:Virginia, ColumnLabel:Beaver, TimeStamp:29827, Value:085713}
```

```
{RowLabel:Virginia, ColumnLabel:Wren, TimeStamp:23136, Value:059359}
{RowLabel:Virginia, ColumnLabel:Zebra, TimeStamp:26189, Value:033051}
{RowLabel:Virginia, ColumnLabel:Zebra, TimeStamp:28412, Value:003787}
{RowLabel:Virginia, ColumnLabel:Zebra, TimeStamp:29191, Value:068444}
{RowLabel:WESTERN_S, ColumnLabel:American_, TimeStamp:22699, Value:017877}
{RowLabel:WESTERN_S, ColumnLabel:American_, TimeStamp:28731, Value:006908}
{RowLabel:WESTERN_S, ColumnLabel:American_, TimeStamp:28885, Value:092084}
{RowLabel:WESTERN_S, ColumnLabel:Anatidae, TimeStamp:19669, Value:073872}
{RowLabel:WESTERN_S, ColumnLabel:Anatidae, TimeStamp:23594, Value:053062}
{RowLabel:WESTERN_S, ColumnLabel:Anatidae, TimeStamp:27747, Value:008511}
```

This test checks that a range row filter and order type 1 is working. Therefore, the results are first ordered in row labels, column labels, and then timestamp. The output above demonstrates that this was successful with row labels “Virginia”, “WESTERN_S alphabetically sorted. Next, within multiple “Virginia”, the column label is sorted next shown by “Baboon”, “Badger”, “Beaver”, etc. Lastly, timestamp is sorted as shown within “Zebra” with timestamps of “26189” first and then “28412” and “29191”. Overall, the results show that both range row filter and order type 1 are working correctly.

Return Records with Single Row Filter and Order Type 3:

query test3 3 Wisconsin * * 2400

```
/home/baani/jdk1.8.0_241/bin/java ...
miniTable>  query test3 3 Wisconsin * * 2400
Replacer: Clock

{RowLabel:Wisconsin, ColumnLabel:Whale_Sha, TimeStamp:10041, Value:046228}
{RowLabel:Wisconsin, ColumnLabel:Whale_Sha, TimeStamp:10188, Value:083033}
{RowLabel:Wisconsin, ColumnLabel:Fish, TimeStamp:10232, Value:050869}
{RowLabel:Wisconsin, ColumnLabel:Goldfinch, TimeStamp:10306, Value:069008}
{RowLabel:Wisconsin, ColumnLabel:Bison, TimeStamp:10602, Value:022386}
{RowLabel:Wisconsin, ColumnLabel:Sheep, TimeStamp:10691, Value:003459}
{RowLabel:Wisconsin, ColumnLabel:Cirrhoscy, TimeStamp:10904, Value:011474}
{RowLabel:Wisconsin, ColumnLabel:Mallard, TimeStamp:10966, Value:045731}
{RowLabel:Wisconsin, ColumnLabel:Lion, TimeStamp:11099, Value:056092}
{RowLabel:Wisconsin, ColumnLabel:Pristioph, TimeStamp:11348, Value:043902}
{RowLabel:Wisconsin, ColumnLabel:Louse, TimeStamp:11366, Value:067111}
{RowLabel:Wisconsin, ColumnLabel:Bison, TimeStamp:11527, Value:093062}
{RowLabel:Wisconsin, ColumnLabel:Pristioph, TimeStamp:11669, Value:002245}
{RowLabel:Wisconsin, ColumnLabel:Carcharhi, TimeStamp:11690, Value:052681}
{RowLabel:Wisconsin, ColumnLabel:Swan, TimeStamp:11709, Value:072168}
{RowLabel:Wisconsin, ColumnLabel:Chiloscyl, TimeStamp:11745, Value:059416}
{RowLabel:Wisconsin, ColumnLabel:Bullhead_, TimeStamp:11959, Value:060605}
{RowLabel:Wisconsin, ColumnLabel:Carcharhi, TimeStamp:12144, Value:058586}
{RowLabel:Wisconsin, ColumnLabel:Pinniped, TimeStamp:12302, Value:009888}
{RowLabel:Wisconsin, ColumnLabel:Sawshark, TimeStamp:12532, Value:072264}
{RowLabel:Wisconsin, ColumnLabel:Apristuru, TimeStamp:12553, Value:016374}
```

```
{RowLabel:Wisconsin, ColumnLabel:Peafowl, TimeStamp:29540, Value:098397}
{RowLabel:Wisconsin, ColumnLabel:Hawk, TimeStamp:29561, Value:020359}
{RowLabel:Wisconsin, ColumnLabel:Gorilla, TimeStamp:29721, Value:064487}
{RowLabel:Wisconsin, ColumnLabel:Columbida, TimeStamp:29989, Value:026431}

======================================

Matched Records: 154
Reads : 2018
Writes: 0

======================================

Total execution time: 0.452 seconds
```

This test checks that a single row filter and order type 3 is working. Therefore, the results are first ordered in row labels and then timestamp. The output above demonstrates that within multiple “Wisconsin” row labels, the value label is then sorted numerically by “12532”, “12553”, etc. Overall, the results show that both single row filter and order type 3 are working correctly.

Return Records with Column Filter and Order Type 4:

query test3 4 * Beaver * 2400

```
miniTable>  query test3 4 * Beaver * 2400
Replacer: Clock

{RowLabel:Tanzania, ColumnLabel:Beaver, TimeStamp:10263, Value:098623}
{RowLabel:WESTERN_S, ColumnLabel:Beaver, TimeStamp:10300, Value:093862}
{RowLabel:Nigeria, ColumnLabel:Beaver, TimeStamp:10534, Value:089766}
{RowLabel:Texas, ColumnLabel:Beaver, TimeStamp:10573, Value:068536}
{RowLabel:Dominica, ColumnLabel:Beaver, TimeStamp:10637, Value:051423}
{RowLabel:New_Jerse, ColumnLabel:Beaver, TimeStamp:10874, Value:080619}
{RowLabel:Slovakia, ColumnLabel:Beaver, TimeStamp:10906, Value:072589}
{RowLabel:Barbados, ColumnLabel:Beaver, TimeStamp:10917, Value:095222}
{RowLabel:Alaska, ColumnLabel:Beaver, TimeStamp:11128, Value:020599}
{RowLabel:Slovenia, ColumnLabel:Beaver, TimeStamp:11243, Value:059540}
{RowLabel:Italy, ColumnLabel:Beaver, TimeStamp:11479, Value:003375}
{RowLabel:Uzbekista, ColumnLabel:Beaver, TimeStamp:11621, Value:031392}
{RowLabel:Malawi, ColumnLabel:Beaver, TimeStamp:11796, Value:083650}
{RowLabel:Ireland, ColumnLabel:Beaver, TimeStamp:11857, Value:096648}
{RowLabel:Slovenia, ColumnLabel:Beaver, TimeStamp:11899, Value:005540}
{RowLabel:Zambia, ColumnLabel:Beaver, TimeStamp:12160, Value:091123}
```

```
{RowLabel:Texas, ColumnLabel:Beaver, TimeStamp:29418, Value:058890}
{RowLabel:Croatia, ColumnLabel:Beaver, TimeStamp:29436, Value:043021}
{RowLabel:Alaska, ColumnLabel:Beaver, TimeStamp:29576, Value:057738}
{RowLabel:Kiribati, ColumnLabel:Beaver, TimeStamp:29600, Value:003112}
{RowLabel:MARSHALL_, ColumnLabel:Beaver, TimeStamp:29687, Value:011015}
{RowLabel:COSTA_RIC, ColumnLabel:Beaver, TimeStamp:29700, Value:091661}
{RowLabel:Maryland, ColumnLabel:Beaver, TimeStamp:29754, Value:073619}
{RowLabel:Virginia, ColumnLabel:Beaver, TimeStamp:29827, Value:085713}

====================================

Matched Records: 184
Reads : 2018
Writes: 0

====================================

Total execution time: 0.167 seconds
```

This test checks that a single column filter and order type 4 is working. Therefore, the results are first ordered in column labels and then timestamp. The output above demonstrates that within multiple “Beaver” column labels, the timestamp label is then sorted numerically by “29418”, “29436”, etc. Overall, the results show that both single column filter and order type 4 are working correctly.

Return Records with Both Row Filter/Column Filter and Order Type 5:

query test3 5 Wisconsin Beaver * 2400

```
miniTable>  query test3 5 Wisconsin Beaver * 2400
Replacer: Clock

{RowLabel:Wisconsin, ColumnLabel:Beaver, TimeStamp:16130, Value:084664}
{RowLabel:Wisconsin, ColumnLabel:Beaver, TimeStamp:22144, Value:080043}
{RowLabel:Wisconsin, ColumnLabel:Beaver, TimeStamp:26508, Value:004993}

=======================================

Matched Records: 3
Reads : 2018
Writes: 0

=======================================

Total execution time: 0.095 seconds
```

This test checks that multiple filters (row and column) as well as order type 5 is working. Therefore, the results are ordered by timestamp. The output above demonstrates that multiple “Wisconsin” row labels and “Beaver” column labels were specifically filtered and retrieved. The timestamp label is then sorted numerically by “16130”, “22144”, “26508” etc. Overall, the results show that both multiple filters and order type 5 are working correctly.

Row Join Tests

- Joining two Tables:

```
rowjoin gan1 gan2 gan3 Zebra 2000
```

```
miniTable>  batchinsert /home/ganesh/Documents/Documents/DBMSI/phase3/test/test/ts1.csv 1 gan1 2000

============================
IndexType 1
============================
{RowLabel:Dominica1, ColumnLabel:Mango, TimeStamp:11, Value:460}
{RowLabel:Dominica1, ColumnLabel:Potato, TimeStamp:12, Value:460}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:1, Value:200}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:5, Value:200}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:13, Value:46067}
{RowLabel:Dominica3, ColumnLabel:Zebra, TimeStamp:10, Value:46063}
{RowLabel:GEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:584, Value:7946}
{RowLabel:Katvia, ColumnLabel:Zebra, TimeStamp:65, Value:48200}
{RowLabel:Liribati, ColumnLabel:Zebra, TimeStamp:352, Value:75102}
{RowLabel:NEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:787, Value:78157}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:653, Value:73324}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:866, Value:54475}
{RowLabel:Romania, ColumnLabel:Zebra, TimeStamp:509, Value:10770}
{RowLabel:South_Carolina, ColumnLabel:Zebra, TimeStamp:266, Value:50897}
{RowLabel:Zimbabwe, ColumnLabel:Zebra, TimeStamp:351, Value:27934}
====================================



miniTable>  batchinsert /home/ganesh/Documents/Documents/DBMSI/phase3/test/test/ts2.csv 1 gan2 2000
============================
IndexType 1
============================
```

```
{RowLabel:South_Carolina, ColumnLabel:Mango, TimeStamp:1, Value:100}
{RowLabel:South_Carolina, ColumnLabel:Mango, TimeStamp:2, Value:300}
{RowLabel:South_Carolina, ColumnLabel:Mango, TimeStamp:4, Value:200}
{RowLabel:South_Carolina, ColumnLabel:Panda, TimeStamp:5, Value:200}
{RowLabel:South_Carolina, ColumnLabel:Zebra, TimeStamp:10, Value:460}
{RowLabel:South_Carolina, ColumnLabel:Zebra, TimeStamp:12, Value:467}
{RowLabel:South_Carolina, ColumnLabel:Zebra, TimeStamp:14, Value:46067}
====================================

miniTable>  rowjoin gan1 gan2 gan3 Zebra 2000

{RowLabel:Dominica1:South_Carolina, ColumnLabel:Dominica1:Mango, TimeStamp:11, Value:460}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:Dominica1:Potato, TimeStamp:12, Value:460}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:South_Carolina:Mango, TimeStamp:1, Value:100}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:South_Carolina:Mango, TimeStamp:2, Value:300}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:South_Carolina:Mango, TimeStamp:4, Value:200}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:South_Carolina:Panda, TimeStamp:5, Value:200}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:Zebra, TimeStamp:12, Value:467}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:Zebra, TimeStamp:13, Value:46067}
{RowLabel:Dominica1:South_Carolina, ColumnLabel:Zebra, TimeStamp:14, Value:46067}
```

This command joins two tables - gan1 and gan2 and stores into new bigtable gan3. We tested this on the column name - “Zebra” and the join results match the expected output. Overall, the results showed the rowjoin functionalities worked correctly.

- Self row join:

```
rowjoin gan2 gan2 gan6 Zebra 2000
```

```
=======================================
miniTable>  batchinsert /home/ganesh/Documents/Documents/DBMSI/phase3/ts2.csv 1 gan2 2000

===========================
IndexType 1
===========================
{RowLabel:Japan, ColumnLabel:Mango, TimeStamp:1, Value:100}
{RowLabel:Japan, ColumnLabel:Mango, TimeStamp:4, Value:200}

{RowLabel:Nepal, ColumnLabel:Mango, TimeStamp:2, Value:300}
{RowLabel:Nepal, ColumnLabel:Panda, TimeStamp:5, Value:200}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:10, Value:460}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:12, Value:467}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:14, Value:46067}

=======================================

miniTable>  rowjoin gan2 gan2 gan6 Zebra 2000

{RowLabel:Nepal:Nepal, ColumnLabel:Nepal:Mango, TimeStamp:2, Value:300}
{RowLabel:Nepal:Nepal, ColumnLabel:Nepal:Panda, TimeStamp:5, Value:200}
{RowLabel:Nepal:Nepal, ColumnLabel:Zebra, TimeStamp:12, Value:467}
{RowLabel:Nepal:Nepal, ColumnLabel:Zebra, TimeStamp:14, Value:46067}

=======================================
```

In Self join, a bigtable is joined to itself. Here, we joined table gan2 to itself on column - “Zebra” and the results match the expected output.

Row Sort Tests

```
rowsort gan1 gan9 Zebra 2000
```

```
miniTable>  batchinsert /home/ganesh/Documents/Documents/DBMSI/phase3/ts1.csv 1 gan1 2000
==========================
IndexType 1
==========================
{RowLabel:Dominica1, ColumnLabel:Mango, TimeStamp:11, Value:460}
{RowLabel:Dominica1, ColumnLabel:Potato, TimeStamp:12, Value:460}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:1, Value:200}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:5, Value:200}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:13, Value:46067}
{RowLabel:Dominica3, ColumnLabel:Zebra, TimeStamp:10, Value:46063}
{RowLabel:GEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:584, Value:7946}
{RowLabel:Katvia, ColumnLabel:Zebra, TimeStamp:65, Value:48200}
{RowLabel:Liribati, ColumnLabel:Zebra, TimeStamp:352, Value:75102}
{RowLabel:NEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:787, Value:78157}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:653, Value:73324}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:866, Value:54475}
{RowLabel:Romania, ColumnLabel:Zebra, TimeStamp:509, Value:10770}
{RowLabel:South_Carolina, ColumnLabel:Zebra, TimeStamp:266, Value:50897}
{RowLabel:Zimbabwe, ColumnLabel:Zebra, TimeStamp:351, Value:27934}

======================================

miniTable>  rowsort gan1 gan9 Zebra 2000
======================================

Row Sort results=>
{RowLabel:Romania, ColumnLabel:Zebra, TimeStamp:509, Value:10770}
{RowLabel:Zimbabwe, ColumnLabel:Zebra, TimeStamp:351, Value:27934}
{RowLabel:Dominica3, ColumnLabel:Zebra, TimeStamp:10, Value:46063}
{RowLabel:Dominica1, ColumnLabel:Mango, TimeStamp:11, Value:460}
{RowLabel:Dominica1, ColumnLabel:Potato, TimeStamp:12, Value:460}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:1, Value:200}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:5, Value:200}
{RowLabel:Dominica1, ColumnLabel:Zebra, TimeStamp:13, Value:46067}
{RowLabel:Katvia, ColumnLabel:Zebra, TimeStamp:65, Value:48200}
{RowLabel:South_Carolina, ColumnLabel:Zebra, TimeStamp:266, Value:50897}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:653, Value:73324}
{RowLabel:Nepal, ColumnLabel:Zebra, TimeStamp:866, Value:54475}
{RowLabel:Liribati, ColumnLabel:Zebra, TimeStamp:352, Value:75102}
{RowLabel:NEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:787, Value:78157}
{RowLabel:GEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:584, Value:7946}
======================================
```

In RowSort tests, we sorted the bigtable on row and column name “Zebra”. We noticed that the results were sorted row wise in lexicological order. The results matched the expected outputs and shows that the map sort operation works correctly.

INTERFACE SPECIFICATIONS

Command Line Interface

There are two commands that the user can execute to interact with the Big Table. They are listed below:

1. Batch Insert
   The command to execute the batch insert is:

   ```
   batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
   ```

batchinsert	command used to execute batch insert
DATAFILENAME	file path of the csv which contains the maps to be inserted
TYPE	integer from 1 to 5 which indicates the type of storage in the BigTable to be used for the maps being inserted
BIGTABLENAME	name of the Big Table that should be created
NUMBUF	Number of buffers to be used for performing batch insert

2. Query
   The command to execute query is

   ```
   query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
   ```

query	Command to execute query
BIGTABLENAME	name of the Big Table that should be created
ORDERTYPE	Integer from 1 to 5 which indicates the type of ordering of the results from the query
ROW FILTER	Row filter

COLUMNFILTER	Column filter
VALUEFILTER	Value filter
NUMBUF	Number of buffers to be used for the query

The filter can be of three types:

I. Matches all values(*)

II A single value. Ex: New_Jersey

III A range specified by two comma separated values in square brackets. Ex [New_Jersey,Serbia]

3. Get Counts

   The command to execute get counts is

   ```
   getCounts NUMBUF
   ```

getCounts	Command to execute get counts
NUMBUF	Number of buffers to be used for getting the counts

4. Row Join

   The command to execute row join is:

   ```
   rowjoin BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER NUMBUF
   ```

getCounts	Command to execute get counts
BTNAME1	Big Table 1
BTNAME2	Big Table 2
OUTBTNAME	New table created to store the results of the join operation
COLUMNFILTER	Column on which we do the join
NUMBUF	Number of buffers to be used for performing the row join operation

5. Row Sort

   The command to perform the row sort operation is:

   ```
   rowsort INBTNAME OUTBTNAME COLUMNNAME NUMBUF
   ```

rowsort	Command to execute get counts
INBTNAME	Input Big table name
OUTBTNAME	Output Big table name where the results are stored
COLUMNNAME	New table created to store the results of the join operation
NUMBUF	Number of buffers to be used for performing the row sortoperation

6. Map Insert
   The command to run map insert is:

```
mapinsert RL CL VAL TS TYPE BIGTABLENAME NUMBUF
```

mapinsert	Command to execute get counts
RL	Row Label of the map to be inserted
CL	Column Label of the map to be inserted
VAL	Value of the map to be inserted
TS	Timestamp of the map to be inserted
TYPE	The storage type into which the map should be inserted
BIGTABLENAME	Bigtable name into which the map should be inserted
NUMBUF	Number of buffers that can be used for the operation

SYSTEM REQUIREMENTS

Operating system: Mac OSX 10.14 Mojave or higher, Ubuntu, Linux
Processor: 2.4 GHz 9th gen Intel Core-i5 or higher
Memory: 8GB or higher
System Type: 64 bit Operating System, x64 bit based processor
Java Version: 1.8.1
Language Level: 8

Executing the Program

1. Navigate to the source directory (`~<PROJECT_DIRECTORY>/MiniTable/src`).
2. Compile the source directory consisting of all related java class programs.

```
<JDKPATH>/bin/javac -classpath <CLASSPATH>
```

3. To execute the program run

```
<JDKPATH>/bin/java cmdline.MiniTable
```

This will provide a command line interface to run the commands for various operations. The various commands which can be used to interact with the big table are discussed in the Command Line Interface section.

Environments

Minibase Implementation is based on Linux based systems. As the project implementation is based on Minibase, it works well on Linux Environments.

Libraries

Minibase Distribution is used as the base Library and no external libraries are used. Only the native java packages are used.

RELATED WORK

This paper [1] introduces bigtable as a new type of distributed storage system. The system is primarily designed to store structured data and allowed scalability on a large scale. This system is used in many services provided by google namely, web indexing, Google Earth, and Google Finance. Each service varies in nature however the bigtable implementation is able to scale and provide a highly flexible approach to a multitude of problems. The bigtable is controlled via an API. The API provides functions for deleting tables and column families. Additionally, bigtable allows the use of regular expressions in order to return subsets of the column families.

Bigtable is designed by indexing a row value, column value and a timestamp. The data model used within the system groups rows into tablets this creates good locality when data is accessed. The columns on the other hands are grouped into column families. The number of distinct column keys is purposely maintained to be small. The final entity that is involved in the indexing is the timestamp, this allows for a history or a log of data. The timestamps allow for multiple entries to be stored, each with different values arranged in descending order, to always return the most recent first.

Bigtable is designed using three major components. These components interact with each other analogously to a master slave architecture. With a single master server and many tablet servers. At the top of the layer exists a Root tablet which consists of all the information needed in order to access each individual tablet. The master is in charge of maintaining all the tablets and tracking the assigned ones and unassigned ones [1].

CONCLUSIONS

In this phase of the project, we have modified our minibase implementation of Bigtable to support multiple index types storage in a single bigtable. We also implemented row join and row sort operations. Implementing join operator helped us understand sort merge join and how it's implemented at a database level. Implementation of row sort operation helped to understand the sort mechanism is existing database management systems with multiple column values. Implementation of multiple batch inserts and sorting and storing the records helped us understand the challenges in its implementation when the same is done on regular databases. The reads and writes are quite less compared to the second phase of the project. The main reason for this is storing data in sorted order always and creation of clustered index on top of it. Though the clustered indexes created are more efficient, the implementation on a regular database where there are lots of reads and writes would be inefficient and costly and is not recommended. We learned a lot of database internals and how each component works, the responsibilities of different components in database management systems, challenges faced and decisions taken when designing the database and it is very crucial as every small decision impacts directly on how performant a database is.

BIBLIOGRAPHY

[1] http://pages.cs.wisc.edu/~dbbook/openAccess/Minibase/
[2] https://cloud.google.com/bigtable/docs/schema-design
[3] https://cloud.google.com/blog/products/gcp/bigquery-under-the-hood

REFERENCES

[1] *Database Management Systems*, R. Ramakrishnan, and J. Gehrke. McGraw-Hill, Third Edition, 2003.
[2] Lecture Notes, *DBMS Implementation*, Dr. Selcuk Candan, Arizona State University, 2020.
[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data.7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), {USENIX} (2006), pp. 205-218.

APPENDIX

Team Member Roles

Team Member Name	Role/Tasks
Kanishk Bashyam	Implementation of distinct row counts, column counts, map counts. Sorting temporary heap files. Integration testing and contributed to the report.
Sumukh Ashwin Kamath	Multiple Batch insert, multiple storage scheme implementation. Map insertion, row join implementation, cli integration, rowsort implementation, debugging, testing and contributed to the report
Baani Khurana	Project management, team management, and communication lead. Implemented shorten string algorithm to limit the max length of characters. Conducted integrated testing of batch inserts, queries, map inserts, row join, and row sort. Structured, organized, and contributed to the report.
Rakesh Ramesh	Multiple Batch insert, multiple storage scheme implementation. Map insertion, Implemented the Evicting queue data structure, Modifying data structure of Map, Standardizing map headers, integration, testing, debugging and contributed to the report.
Sreenivasan Ramesh	Implementation of row sort, row join and command-line interface. Helped with integration, debugging and testing of row sort, row join, batch insert and writing the report.
Ganesh Ashok Yallankar	Row join implementation and command-line interface. Map insertion command-line interface. Stream modification for multiple index types. Testing, debugging and contributed to the report.