CSE 510 - Database Management System Implementation

Spring 2020

Phase III

Group Number #2

Group Members:
Kshitij Dhyani
Aashka Bhavesh Parikh
Maneesha Guntur
Himali Hemant Gajare
Rigved Alankar

Abstract

Minibase was originally implemented as an RDBMS. The objective of the phase 3 report is to describe the steps involved in using Minibase modules to implement a Big table like DBMS. To achieve this, we modified the tuple-based implementations in Minibase into map-based implementations. More specifically, this phase involves additional functionalities such as multiple batch inserts, row join operations, row sort operations, individual map inserts etc. Additionally, there also exists a comprehensive command line interface that allows for retrieving read/write counts, querying and performing all other operations seamlessly through the means of the terminal. Lastly, the report involves an output and analysis section, that runs through all the above operations by varying buffer size, index types, query types etc and tracks the number of reads and writes count.

Keywords

Java, Minibase, Buffer Management, IndexScan, FileScan, Disk Space Management, Heap Files, B-trees, Index, Joins, Sort, Makefile, Query, DBMS, RDBMS, BigTable, Map, Batch Insert, Row Label, Column Label, Timestamp, Value, Stream, Read, Write, Counter, Clustering, Key-Value, IndexFiles, Row Join, Row Sort, Map Insert, Batch Insert

Table of Contents

Introduction

Big tables are a great leap forward in the direction of high-volume and scalable database solutions. As the needs of our data-intensive world grow with time, transitioning to scalable solutions such as Big Tables becomes a necessity. Moreover, concepts like distributed storage, indexing, joining and sorting become even more important. These concepts allow for faster querying, and optimized query costing, and maximizes performance.

As the world leans towards cloud storage and cloud solutions, we are bound to see big table implementations all across the DBMS domain.

Goals

During phase 2 we transitioned from a tuple-based RDBMS type DB to a Bigtable-like DB. Through phase 3, our main objective is to further enhance and propel the capabilities of our Bigtable-like DB by adding new features such as implementing row join and row sorting. Moreover, adding a feature to maintain multiple index types, storage schemes etc. Additionally, we aim to allow capabilities of batch and individual map insertion into the DB. Lastly, we aim to implement a read-write count generator and use it to track and analyse behaviour or all our implemented feature. In order to achieve these endeavours, we made changes to the disk manager module, iterator package, MainTest.java file for command lines interface changes etc.

Terminology

Database:

The concept of a database refers to a grouping of related observations, comprising a set of data that is recorded and structured in a manner that facilitates efficient retrieval. To formally describe the storage mechanisms for a database, various data models are employed, including physical models such as data structures, logical models such as relational, OO, and OR, and conceptual models such as UML, ER, and Extended ER. These models are subject to tradeoffs in terms of performance and expressive power [4].

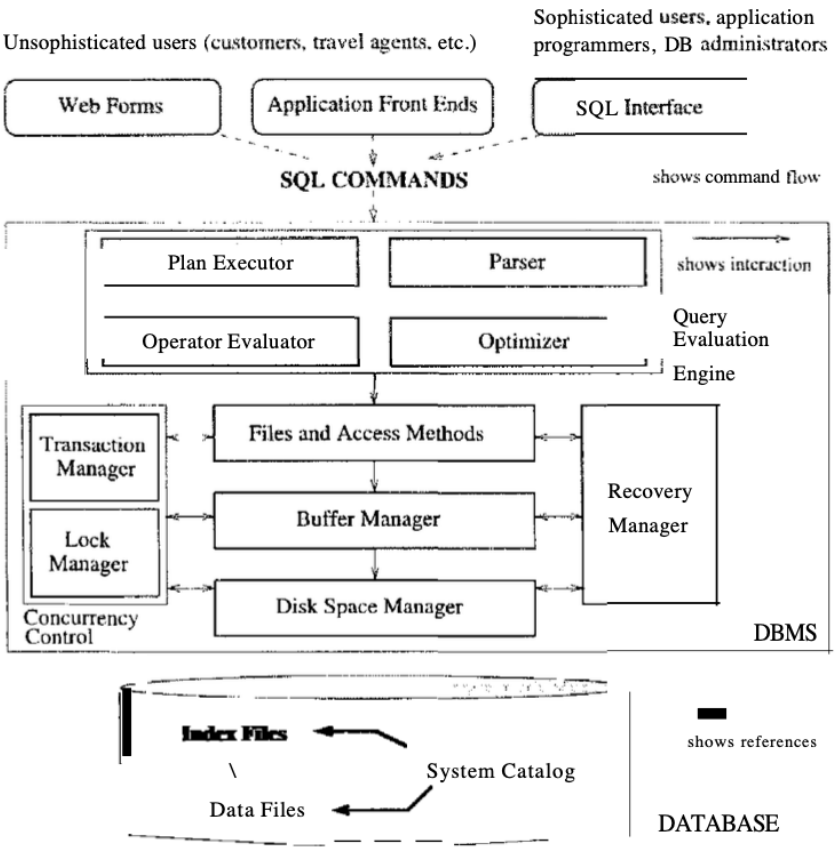

Figure 1. Architecture of DBMS

Maps:

Maps is a key-value pair type of datasets that enable associating data values to a set key. The set keys can be used for identifying the data values. With reference to big table architecture within the Java Minibase, Maps have a combination of 3 keys that point to a single String value.

```
(row : string, column : string, time : int) → string.
```

B-Trees:

B-Trees are a specialization of m-way trees that can balance themselves [2].

B+ Trees:

B+ Trees are also an advanced variation of self-balancing trees, where all data points are leaf nodes [2]. The B+ tree is a highly utilized index in modern database systems. Each node in the tree, represented as a disk page, typically has a size of

4096 bytes. The leaves of the tree hold the database records, while index pages located above the leaf level contain only index records. The search for a single database record involves accessing exactly one node at each level of the tree. Due to its organization of clustering data in leaf pages in the order of the indexing attributes, the B+ tree is optimal for key-range searches. Additionally, the B+ tree is dynamically organized, ensuring that the height of the tree remains balanced even after record insertions or deletions [3].

Big Tables:

Big Tables are wide-column databases with a key-value (NoSQL) like implementation capable of handling large volumes of data often in the size of petabytes [3].

Joins:

Join operations are effectively what their name specifies, they try to combine all the related rows across different tables, given a condition. Different join algorithms are applicable in different scenarios such as Simple Nested Loop Joins, Page Nested Loop Joins, Index Nested Loop Joins, Sort Merge Joins, Hash Joins etc. The two joins we have implemented are Nested Loop Join and Sort-Merge Join.

- Nested Loop Join: It works by iterating through each record in the outer table and then iterating through each record in the inner table to check for a match. This process is repeated for each row in the outer table, and the matching records are combined to produce the result set.
- Sort-Merge Join: This algorithm requires that the input tables be sorted on the join key, and works by scanning through the sorted tables sequentially and matching the rows with the same join key value. The algorithm involves three main steps. First, the input tables are sorted based on the join key. Second, the sorted tables are merged by comparing the values of the join key. Finally, a new table is created that contains the matched rows from the two input tables.

Nested Loop Join is considered the simplest join algorithm and can perform well for small to medium-sized tables. Sort Merge Join is commonly used for large tables as it can handle data that does not fit in memory. However, it may have a higher overhead due to the sorting process.

Sort:

Sorting is a way of ordering/arranging data such that it adds meaning to it. Sorting proves to be an integral part of a DBMS since its plays an important role in optimizing query results, indexing, joins etc.

Indexing:
It is a technique that reduces disk operations such as read, and write operations in a database management system [4].

Minibase:
Minibase is a database management system specially designed to aid students in the learning of DBMS concepts and their inner workings. All DBMS functionalities are implemented as Tuple constructs [5].

Heap File:
Heap file is a type of file organization used to store data in a database management system (DBMS). In a heap file, data is stored in a collection of pages, and the order of the pages is unimportant. This means that there is no fixed order or structure to the data stored in a heap file, and data can be inserted, deleted, or updated at any position within the file [4].

Assumptions

1. The RowLabel, Column Label, and Value fields are categorized as 'String' (Java.lang.String).
2. The TimeStamp field is assumed to be of Integer type.
3. The default length of the RowLabel, Column Label, and Value fields is 20 bytes. If the length of any field exceeds it will throw an error.
4. If the size of the Map exceeds the page size(1024 Bytes) it will throw an error.
5. If the program is terminated and restarted, the previously inserted records must be re-inserted since there is no cache maintained for them.
6. It is assumed that The Range filters are closely packed in the query, without any whitespace between the range values.
7. When we join 2 tables, the output table will not be stored.

Implementation

The section provides an in-depth explanation of the functionalities that were incorporated in the phase 3 of the project.

Task 1: Implement a command line GetCount functionality

The command line interface for get count functionality is as follows:

```
getCounts Bigtablename NUMBUF
```

Let us breakdown the functionality of each identifier in the command

getCounts - This acts as an identifier, to allow the interpreter to recognise the request for a getCounts operation.

NUMBUF - represents the number of buffers that will be used in the operation.

In order to achieve this functionality, we created a new class called GetAllCount.
BigT - GetAllCount
The get count of all maps present in the bigtable. It will computes the number of maps in any given big table, along with the number of distinct columns and rows.

The different constructors:

1. `GetAllCount(int numbuf)`: Sets the number of buffer as per the user input.

Finally, the methods mentioned below are used to interact with the map and perform different operations.

`void run(byte[] mapPtr)`: Takes the name of the big table as an input and creates a bigt instance

Task 2: Modify BigDB class and bigT class

In this task we are equipped with the task of modifying the BigDB and bigt class so that they don't take a type parameter but we will mimic a distributed database system which stores a DB across 5 heap files which are in sync with each other. The storage mechanism is defined by the following order -

– Type 1: No index

– Type 2: one btree to index row labels

– Type 3: one btree to index column labels

– Type 4: one btree to index column label and row label (combined key)

– Type 5: one btree to index row label and value (combined key)

Task 3: Modify BatchInsert program

In Phase 3, we are supposed to modify batch insert command in such a way that it is possible to insert multiple batches of records. If a duplicate record is found during insertion (the record must have the same row label, column label, and timestamp) then it is not inserted into the big table. A HashMap maintains the version of all the records inserted and at a time keeps only three versions of a particular row label column label combination. The command should also support inserts into an existing table across different indices. By doing so, we allow multiple index creation on different sets of records within the big table. However, we modify the Stream function in a way such that it reads data from all five different heapfiles instead of reading only a single file. This helps us implement and test the true distributed nature of bigtables.

The command line invocation for multiple batch insert is as follows:

batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF

Let us breakdown the functionality of each identifier in the command

batchinsert: This acts as an identifier, to allow the interpreter to recognise the request for a batch insert operation.

DATAFILENAME: This acts as an identifier that allows the interpreter to recognised the storage location of the csv file that contains the data values to be inserted.

BIGTABLENAME: This specifies the name of the bigtable that is to be created after the batch insert query is completed.

TYPE**:** This acts as an identifier to the interpreter, allowing it to recognize the type of indexing and clustering method as discussed above. There can be 5 different types of indexing and clustering methods, each identified with the corresponding integer value.

NUMBUF: This represents the number of buffer pages, that will be involved in the execution of the batch insert operation.

To implement these changes, we primarily modify two components of the bigt:

1. `MID insertMap(byte[] mapPtr)`**:** This function is updated to maintain the HashMap for the versions of different maps.
2. `Stream openStream(String bigTableName, int type, int orderType, String rowFilter, String columnFilter,String valueFilter, int numBuf)` **:** This function is updated such that it does not contain the storageType anymore. Regardless of the type of the batchinsert commands, the openStream would read all the heap files to generate an output.

Task 4: Implement Map Insert functionality

The command line interface for map insert functionality is as follows:

mapinsert RL CL VAL TS TYPE BIGTABLENAME NUMBUF

Let us breakdown the functionality of each identifier in the command -
RL: Row Label of the Map
CL: Column Label of the Map
VAL: Value to be inserted
TS: Timestamp

Type: It is an integer which lies between 1 and 5 which indicates the storage type of the map

BIGTABLENAME - name of the bigtable that will be created in the database. If the bigtable already exists in the database, the map will be inserted into the existing bigtable

NUMBUF - Number of buffers used in the operation

To implement these changes, we created a new class within BigT with the following functions.

<u>BigT - MapInsert</u>
The map insert has one function called run which implements

The different constructors:

1. `MapInsert(String bigTable, int storageType, bigt table, Version r)`: Sets the bigtable, storage type, table and the version as per the user's input for the referenced MapInsert object.

Finally, the methods mentioned below are used to interact with the map and perform different operations.

`HashMap<String, List<Version>> run (String RowLabel, String ColumnLabel, String Value, int TimeStamp, HashMap<String, List<Version>> mapVersion)`: This function uses the mapinsert object and create a map out of the rowlabel, columnlabel, value and timestamp fields and executes an insertMap function on the bigt instance to push the map into the table.

Task 5: Implement program query

As compared to phase 2, there is a slight change in the functionality of the query, it does not require "type" as input since now it would traverse across all the heapfiles for the corresponding bigtable to fetch the results. This would help in showcasing the distributed nature of the bigtable.

The command line interface for map insert functionality is as follows:

query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF

QUERY: This is a token. It acts as an identifier to the interpreter, allowing it to recognize the command as a query command.

BIGTABLENAME: This specifies the name of the BigTable on which we are to perform the query.

ORDERTYPE: This acts as an identifier for the type of ordering of the result of the query. There are 5 different types of order types defined in the programming of the big tables. They are represented by their corresponding integer values.

1 - Ordered in the following priority order: Row Label -> Column Label -> Time Stamp
2 - Ordered in the following priority order: Column Label -> Row Label -> Time Stamp
3 - Ordered in the following priority order: Row Label -> Time Stamp
4 - Ordered in the following priority order: Column Label -> Time Stamp

5 - Ordered in the following priority order: Time Stamp

FILTERS: There are 3 possible filters in the query, namely - Row, Column and Value Filter. Each filter can then take the following 3 forms:

1) Match All values - represented by a * operator
2) Match a single Value - represented by mentioning the value - Example: “Switzerland”
3) Matching to a group of values - represented using comma separated values enclosed in a pair of square brackets - Example: [“Sweden, Canada”]

NUMBUF: This represents the number of buffer pages that the query will engage in its executions.

Task 6: Implement BigT Join Operator

The goal of this operation is to implement two different joining strategies for two tables in the bigt environment. The output of the join should be such that, a collection of maps consisting of matching records are stored in a new bigtable. The matching criteria are such that, two rows match only if they have the same column and the most recent values for the two columns are the same. We store two maps for cases like this, with column label _left and _right depending on the position of the relation. The idea is to be able to identify where the map is coming from. In both the records the rowlabel is consistent which is “rowlabel (left relation) : rowlabel (right relation)”. Another criterion for joining is that if the values match in both relations, we output both the records with the original column label but modified row label “rowlabel (left relation) : rowlabel (right relation)”.

In the project we have used two different joining strategies -

i) Simple Nested Join Loop Algorithm - In Simple Nested Join Loop Algorithm, we sequentially go through every row of the first table matching its contents by traversing the entire contents of the second table. The algorithm of the join is as follows -

```
For every row ri in table 1:
    For every row sj in table 2:
            If ri == sj:
                yield<ri,sj>
```

ii) Sort Merge Join - Sort Merge Join consists of two phases -

1) Sort - Involves sorting the relations on the join key
2) Merge - Scan the sorted relations and find matching tuples.

Since all the rows are sorted, rather than traversing through the entire table we can start from the last row in the inner table where the match was found thus reducing the time complexity.

To implement the BigT join operator we have created a new RowJoin class which has the following constructors and functionalities:

The constructors of the class:

1. `public RowJoin(String bigTable1, String bigTable2, String columnName, String outBigTable, int joinType, int numBuf):` This constructor initialises two temporary heapfiles, joining type, number of buffers and the output big table. These temporary heapfiles are created and destroyed every time a join is called. One heapfile maintains only the most recent maps for all the row label column label combinations for the outer relation. Whereas the other heapfile does the same for the inner relation. The join functions use only these files to build the final output.

The functions in the RowJoin class are

1. `HashMap<String, List<Version>> run(HashMap<String, List<Version>> outputMapVersion):` This function is called from the MainTest classes using the RowJoin object. It identifies the join type from the object and accordingly executes either performNestedLoopJoin() or performSMJoin()

2. `HashMap<String,List<Version>> performJoinNestedLoop(HashMap<String,List<Version>> outputMapVersion)`: Within this function, two streams have been set up on the temporary heap files and they have been iteratively merged as discussed in the Nested Loop algorithm.
3. `HashMap<String,List<Version>> performSMJoin(HashMap<String, List<Version>> outputMapVersion)`: This function creates an object of SortMergeMap which is explained below. Using the object a performJoin2 operation is called which is in the SortMergeMap class, which merges the relations using the sort-merge strategy.
4. `void buildTempHeapFiles()`: Opens a sorted scan on the two relations to generate the data for the two temporary heapfiles. The temporary heapfiles will only have the latest map for any rowlabel, columnlabel combination.
5. `CondExpr[] getConditionalExpression(String valueLabel)`: Generates a CondExpr list for any given value filter.

As discussed above, we have also implemented a new SortMergeMap class which has the entire logic behind the sort-merge algorithm.

The class has the following constructors and functions.

1. `SortMergeMap(String bnames1, String bnames2, FileScanMap s1, FileScanMap s2, String outBigTableName)`: This takes the name of the two relations, along with their FileScanMap object which was initialised in the RowJoin class and the name of the output bigtable. In this case the name of two input relations would be the name of the two temporary heap files.

It has the following functions as well.

1. `HashMap<String, List<Version>> performJoin2(HashMap<String, List<Version>> outputMapVersion)`: This function uses the two scan operators along with one List to iterate over the two relations along with managing duplicates to implement the sort merge algorithm.

Task 7: Implement Command Line Program Join

The command line interface for the functionality is given below -
rowjoin BTNAME1 BTNAME2 OUTBTNAME JOINTYPE NUMBUF COLUMNFILTER

Let us breakdown the functionality of each identifier in the command -
BTNAME1 - Name of the big table on the left
BTNAME2 - Name of the big table on the right
OUTBTNAME - Name of the big table in which the output will be stored
JOINTYPE - Indicates the type of join we will be performing. In the project we have implemented two join algorithms -
i) Nested Loop Join
ii) Sort Merge Join
NUMBUF - Number of buffers used in the operation
COLUMNFILTER - Column filter

Task 8: Implement Command Line Program RowSort

The command line interface for the functionality is given below -
rowsort INBTNAME OUTBTNAME COLUMNNAME NUMBUF
Let us breakdown the functionality of each identifier in the command -
INBTNAME - Input Big Table name
OUTBTNAME - Output Big Table name
COLUMNNAME - Column name
NUMBUF - Number of buffers used in the operation

In order to achieve this functionality, we created a new class called RowSort under the BigT package. The main intention of row sort is to sort the rows of the source table and store them in a separate table based on the row order provided by the user.

The constructor of the class is

```
RowSort(String sourceTableName, String resultTable, int
rowOrder, bigt bt): Sets the source table, resulting table,
row order and the big table as per the user input.
```

Finally, the methods mentioned below are used to interact with the map and perform different operations.

`void run(HashMap<String, List<Version>> mapVersion)`: Takes the versions of the maps of the big table as an input and then creates two Stream objects for scanning the table. One Stream object will be used for the filtering column label based on the the order type(sorted on timestamp), while the other steam object will be used for retrieving all the records (sorted based on the row and column.

Task 9: Implement Command Line Program to create indexes

The command line interface for the functionality is given below -

createindex BTNAME TYPE

Let us breakdown the functionality of each identifier in the command -
BTNAME - Big Table name
TYPE - Indicates the storage type

In order to achieve this functionality, we created a new class called CreateIndex under the BigT package.
The create index has one function which changes the index of a map from the current index type to the new index type provided as a parameter by the user.

The constructor of the class is:

> `create Index(String btName, int old_indextype, int new_indextype, bigt bt)`: Sets the big table name, old index type, new index type and the big table as per the user input.

Finally, the methods mentioned below are used to interact with the map and perform different operations.

> `void createIndex(HashMap<String, HashMap> tableVersions)`: This method will fetch the records from the hashmap(containing mapversions) and then will create new index on the new_indextype.

Interface Specification

A command Line Interface was developed by the team to facilitate interaction with the Big Tables. It supports the following functionalities:

i) Batch Insert
ii) Query
iii) Get Counts
iv) Map Insert
v) Row Sort
vi) Creating Index

System Requirement

Processor: 2.3 GHz Quad-Core Intel Core i5 or Higher
Graphics: Intel Iris Plus Graphics or Higher
Memory: 8 GB 2133 MHz LPDDR3 or Higher
MacOS: Ventura 13.2, or any other Linux Distros
Java "15.0.2" 2021-01-19
Java(TM) SE Runtime Environment (build 19.0.2+7-44)
Java HotSpot(TM) 64-Bit Server VM (build 19.0.2+7-44, mixed mode, sharing)

Execution

1. Navigate to the src directory which is given by the following path -

    <PROJECT_DIRECTORY>/minjava/javaminibase/src

2. Using the following command compile the relevant java classes in the directory.

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

3. The following command executes the main program -

    <JDKPATH>/bin/java tests.MainTest

After executing the above command, the user is presented with a console-based menu which consists of the following options -

Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin

Press 9 to Quit

Pressing 1 allows the user to batchinsert the data in the bigtable while pressing 2 will allow the user to query a bigtable based on the filters,index and order types. Pressing 3 will result in the termination of the application.

Environment

Linux based running environments are preferred, since Minibase has been originally implemented around Linux based systems and distros. As a team we personally worked on the following operating systems: MacOS: Ventura 13.2, Ubuntu 22.04 etc.

Libraries

No use of any external libraries. The library suite for this project solely comprises the Minibase Distribution.

Related Work

The paper titled “Bigtable: A Distributed Storage System for Structured Data” introduces the concept of big tables [4], and walks us through a distributed storage system, capable of handling petabytes of data, which has reasonable flexibility and scalability. Big tables do not offer relational models; rather it emphasizes dynamic control over data layouts and formats. Big tables allow clients of the database to have certain decision power over the location of their stored data, and to improve data access and manipulations, indexes are produced with respect to column and row labels.

Output and Analysis

When we run the getCounts command, it will retrieve the number of maps present in the table.

```
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------------
3
FORMAT: getCounts BIGTABLENAME NUMBUF
getCounts flop 100
Bigtable: flop
--------------------------------------
TOTAL NUMBER OF MAPS: 6640
NUMBER OF DISTINCT ROW LABELS: 51
NUMBER OF DISTINCT COLUMN LABELS: 51
--------------------------------------
TIME TAKEN 3 s
------------------------ Counts --------------------------------
READ COUNT : 2646
WRITE COUNT : 1248
------------------------------------------------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------------
3
FORMAT: getCounts BIGTABLENAME NUMBUF
getCounts new_flop 100
Bigtable: new_flop
--------------------------------------
TOTAL NUMBER OF MAPS: 1
NUMBER OF DISTINCT ROW LABELS: 1
NUMBER OF DISTINCT COLUMN LABELS: 1
--------------------------------------
TIME TAKEN 0 s
------------------------ Counts --------------------------------
READ COUNT : 21
WRITE COUNT : 0
------------------------------------------------------------------
```

When we insert record into the bigtable using the mapinsert into existing table.

```
------------------------------------------------------------------
4
FORMAT: mapinsert ROWLABEL COLUMNLABEL VALUE TIMESTAMP TYPE BIGTABLENAME NUMBUF
mapinsert sanfrancisco lion 100 10000 2 flop 400
Total number of maps: 6640

------------------------ Counts --------------------------------
READ COUNT : 427
WRITE COUNT : 25
------------------------------------------------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------------
2
FORMAT: query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query flop 1 * lion * 400
[sanfrancisco lion 10000 ] -> 100
Time Taken 50 ms
Record Count : 1

------------------------------------------------------------------


Distinct rows : 51
Distinct columns : 51

------------------------ Counts --------------------------------
READ COUNT : 3213
WRITE COUNT : 1249
------------------------------------------------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------------
```

When we insert record into the bigtable using the mapinsert into new table. It will create a new table and insert a record into it.

```
------------------------------------------------------------
4
FORMAT: mapinsert ROWLABEL COLUMNLABEL VALUE TIMESTAMP TYPE BIGTABLENAME NUMBUF
mapinsert sanfrancisco lion 10 100 2 new_flop 400
Total number of maps: 1

------------------------ Counts ------------------------------
READ COUNT : 15
WRITE COUNT : 0
------------------------------------------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------
2
FORMAT: query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query new_flop 1 * * * 400
[sanfrancisco lion 100 ] -> 10
Time Taken 25 ms
Record Count : 1

------------------------------------------------------------


Distinct rows : 1
Distinct columns : 1

------------------------ Counts ------------------------------
READ COUNT : 21
WRITE COUNT : 0
------------------------------------------------------------
```

We can create multiple indexes on the table.

```
------------------------------------------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------
6
FORMAT: createindex BIGTABLENAME ORIGINAL_STORAGE_TYPE NEW_INDEX_TYPE
createindex flop 2 3
```

```
[Zimbabwe Raven 34054 ] -> 5049
[Zimbabwe Reindeer 3320 ] -> 1486
[Zimbabwe Reindeer 22866 ] -> 5324
[Zimbabwe Reindeer 28556 ] -> 5527
[Zimbabwe Rhinocero 56360 ] -> 4603
[Zimbabwe Rhinocero 86989 ] -> 6891
[Zimbabwe Rhinocero 98407 ] -> 5289
[Zimbabwe Sheep 67617 ] -> 6445
[Zimbabwe Skunk 38666 ] -> 1847
[Zimbabwe Skunk 61855 ] -> 6706
[Zimbabwe Skunk 62990 ] -> 8314
[Zimbabwe Swallow 16207 ] -> 1227
[Zimbabwe Swallow 77880 ] -> 881
[Zimbabwe Wren 77494 ] -> 5819
[Zimbabwe Wren 83368 ] -> 6591
[Zimbabwe Zebra 35416 ] -> 2303
TIME TAKEN 4 s
----------------------- Counts ------------------------------
READ COUNT : 3110
WRITE COUNT : 2619
-------------------------------------------------------------
```

If we create an index on index 5 the read and write counts are slightly greater than previous because the index key is larger than previous.

```
[Zimbabwe Rhinocero 98407 ] -> 5289
[Zimbabwe Sheep 67617 ] -> 6445
[Zimbabwe Skunk 38666 ] -> 1847
[Zimbabwe Skunk 61855 ] -> 6706
[Zimbabwe Skunk 62990 ] -> 8314
[Zimbabwe Swallow 16207 ] -> 1227
[Zimbabwe Swallow 77880 ] -> 881
[Zimbabwe Wren 77494 ] -> 5819
[Zimbabwe Wren 83368 ] -> 6591
[Zimbabwe Zebra 35416 ] -> 2303
TIME TAKEN 4 s
----------------------- Counts ------------------------------
READ COUNT : 3466
WRITE COUNT : 2679
-------------------------------------------------------------
```

When we run the rowsort command it will display the following output. The maps containing the column filter will consist of the records having column value equal to column filter and it will be sorted based on the timestamp and row. While in other set in which it does not contain column filter will be sorted on only on row and column.

```
------------------------------------------------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for getCounts
Press 4 for MapInsert
Press 5 for rowSort
Press 6 for createindex
Press 7 for rowjoin
Press 9 to Quit
------------------------------------------------------------------
5
FORMAT: rowsort INBTNAME OUTBTNAME ROWORDER COLUMNNAME NUMBUF
ROWORDER:
 1. Ascending
 2. Descending
rowsort flop flop_new 1 Magpie 400
```

```
[Zimbabwe Partridge 91377 ] -> 9426
[Zimbabwe Partridge 89143 ] -> 6500
[Zimbabwe Peafowl 47596 ] -> 2155
[Zimbabwe Peafowl 80375 ] -> 9671
[Zimbabwe Peafowl 87870 ] -> 4581
[Zimbabwe Pelican 6646 ] -> 6498
[Zimbabwe Pelican 47141 ] -> 3839
[Zimbabwe Pelican 15053 ] -> 7647
[Zimbabwe Pinniped 79555 ] -> 1493
[Zimbabwe Quail 98161 ] -> 9477
[Zimbabwe Quail 4522 ] -> 7336
[Zimbabwe Raven 34054 ] -> 5049
[Zimbabwe Raven 20795 ] -> 5277
[Zimbabwe Reindeer 28556 ] -> 5527
[Zimbabwe Reindeer 3320 ] -> 1486
[Zimbabwe Reindeer 22866 ] -> 5324
[Zimbabwe Rhinocero 86989 ] -> 6891
[Zimbabwe Rhinocero 56360 ] -> 4603
[Zimbabwe Rhinocero 98407 ] -> 5289
[Zimbabwe Sheep 67617 ] -> 6445
[Zimbabwe Skunk 38666 ] -> 1847
[Zimbabwe Skunk 61855 ] -> 6706
[Zimbabwe Skunk 62990 ] -> 8314
[Zimbabwe Swallow 77880 ] -> 881
[Zimbabwe Swallow 16207 ] -> 1227
[Zimbabwe Wren 83368 ] -> 6591
[Zimbabwe Wren 77494 ] -> 5819
[Zimbabwe Zebra 35416 ] -> 2303


Displaying Maps containing the column
[Australia Magpie 72354 ] -> 4641
[COSTA_RIC Magpie 73737 ] -> 4228
[COSTA_RIC Magpie 76534 ] -> 5876
[COSTA_RIC Magpie 94948 ] -> 5405
[CZECH_REP Magpie 25519 ] -> 9967
[CZECH_REP Magpie 41797 ] -> 9394
[CZECH_REP Magpie 44278 ] -> 8989
[Canada Magpie 55759 ] -> 3195
[Canada Magpie 57287 ] -> 1665
[Canada Magpie 74569 ] -> 6667
[Colombia Magpie 55967 ] -> 6402
[Croatia Magpie 77331 ] -> 3472
[Croatia Magpie 91180 ] -> 934
[Croatia Magpie 98310 ] -> 4556
[Cyprus Magpie 21691 ] -> 6659
[Cyprus Magpie 38767 ] -> 9324
```

Result when joining the two tables using simple nested loop

```
2
FORMAT: query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query trial1 1 * * * 400
[Malaysia:Malaysia Peafowl_left 33204 ] -> 9681
[Malaysia:Malaysia Peafowl_right 33204 ] -> 9681
[NEW_ZEALA:NEW_ZEALA Peafowl_left 4237 ] -> 9537
[NEW_ZEALA:NEW_ZEALA Peafowl_right 4237 ] -> 9537
[Nigeria:Nigeria Peafowl_left 22807 ] -> 9395
[Nigeria:Nigeria Peafowl_right 22807 ] -> 9395
[Saudi_Ara:Saudi_Ara Peafowl_left 39036 ] -> 9396
[Saudi_Ara:Saudi_Ara Peafowl_right 39036 ] -> 9396
[Sweden:Sweden Peafowl_left 4050 ] -> 9863
[Sweden:Sweden Peafowl_right 4050 ] -> 9863
[TRINIDAD_:TRINIDAD_ Peafowl_left 23790 ] -> 9618
[TRINIDAD_:TRINIDAD_ Peafowl_right 23790 ] -> 9618
[Tanzania:Tanzania Peafowl_left 84607 ] -> 9388
[Tanzania:Tanzania Peafowl_right 84607 ] -> 9388
[Tonga:Tonga Peafowl_left 82800 ] -> 9650
[Tonga:Tonga Peafowl_right 82800 ] -> 9650
[Zimbabwe:Zimbabwe Peafowl_left 80375 ] -> 9671
[Zimbabwe:Zimbabwe Peafowl_right 80375 ] -> 9671
Time Taken 21 ms
Record Count : 18
```

```
rowjoin left right trial1 1 400 Peafowl
Records in outer relation: 10
Records in inner relation: 10
--------------Get Map Count: 10
--------------Get Map Count: 9
TIME TAKEN 0 s
------------------------- Counts ---------------
READ COUNT : 6
WRITE COUNT : 1
------------------------------------------------
```

Result when joining the two tables using merge sort algorithm.

```
2
FORMAT: query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query trial2 1 * * * 400
[Malaysia:Malaysia Peafowl_left 33204 ] -> 9681
[Malaysia:Malaysia Peafowl_right 33204 ] -> 9681
[NEW_ZEALA:NEW_ZEALA Peafowl_left 4237 ] -> 9537
[NEW_ZEALA:NEW_ZEALA Peafowl_right 4237 ] -> 9537
[Nigeria:Nigeria Peafowl_left 22807 ] -> 9395
[Nigeria:Nigeria Peafowl_right 22807 ] -> 9395
[Saudi_Ara:Saudi_Ara Peafowl_left 39036 ] -> 9396
[Saudi_Ara:Saudi_Ara Peafowl_right 39036 ] -> 9396
[Sweden:Sweden Peafowl_left 4050 ] -> 9863
[Sweden:Sweden Peafowl_right 4050 ] -> 9863
[TRINIDAD_:TRINIDAD_ Peafowl_left 23790 ] -> 9618
[TRINIDAD_:TRINIDAD_ Peafowl_right 23790 ] -> 9618
[Tanzania:Tanzania Peafowl_left 84607 ] -> 9388
[Tanzania:Tanzania Peafowl_right 84607 ] -> 9388
[Tonga:Tonga Peafowl_left 82800 ] -> 9650
[Tonga:Tonga Peafowl_right 82800 ] -> 9650
[Zimbabwe:Zimbabwe Peafowl_left 80375 ] -> 9671
[Zimbabwe:Zimbabwe Peafowl_right 80375 ] -> 9671
Time Taken 17 ms
Record Count : 18
```

Conclusions

Through this project, we transitioned a relational DBMS (Java Minibase) into a bigtable-like DBMS, and added new features to its existing architecture. Playing around with the code base and making improvements and adding new functionalities to the existing code gave us confidence in the concepts, and opened the door to a deeper level of understanding. Additionally, implementing distributed data systems and varied storage schemes gave us insights into how industry cloud solutions work. Implementing row joins and row sorts gave us a profound understanding of underlying algorithms that run a Database management system. Through this course we made granular changes to the code base and played around with the query system, this has made us thoroughly familiar with concepts such as big tables, indexing and joins, distributed data systems etc, and has elevated our understanding of the DBMS domain.

Bibliography

[1] https://www.scylladb.com/glossary/wide-column-database/
[2] https://www.geeksforgeeks.org/difference-between-b-tree-and-b-tree/
[3] https://en.wikipedia.org/wiki/Bigtable
[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM *Transactions on Computer Systems* (TOCS), 26(2), 1-26.

Appendix

Team Member Roles

Team Member Name	Roles and Responsibilities
Kshitij Dhyani	Primary contributor towards query manipulation, removal of types, and map insert. Contributed abundantly towards the report and testing and debugging of the code base.
Aashka Bhavesh Parikh	Implemented row join and contributed to the implementation of row sort. Helped in code integration, testing and debugging. Contributed to report structure and research.
Maneesha Guntur	Primary contributor towards the theory and understanding of Sort merge join and nested loop join. Largely helped with the implementation of distributed heap files. Worked extensively on the report structure and project organization.
Himali Hemant Gajare	Worked extensively on index creation and rowSort implementation. Governed and led the direction of the project and largely helped with code versioning, and integrated different code branches.
Rigved Alankar	Implemented Batch Insert and contributed to implementing getCounts. Worked with analysis of different operations, and contributed extensively to report structure and formatting.