ARIZONA STATE UNIVERSITY

CSE 510: Database Management Systems Implementation

Spring 2020

Project Phase 3 – **“Implementing BigDB”**

Group Members

Vineeth Sai Surya Chavatapalli

Gnana Prakash Avvaru

Arun Raj Deva

Kavya Sree Rajula

Sanjana Dasari

Lokeshwar Dasa

TABLE OF CONTENTS

ABSTRACT

The objective of the project is to implement a Bigtable-like DBMS using different components of MiniBase. The goal of this phase is to extend the batch insertion, query and sorting functionalities of phase 2 and implement map insert, query, row join and row sort operations. Querying with equality search and range search on indexes helps us to improve the processing time of the search queries.

1 INTRODUCTION

1.1 Terminology

1) Map: A Map is a data structure which contains attributes that make up a relation. A Map contains 4 attributes which are row label, column label, timestamp, value.

2) Index: An index is a set of ordered references to rows of a table. An index is a small table which has only two columns. The first column consists of a copy of the primary key or other keys and the second column consists of a pointer which holds the address of a particular map. Index can improve the query execution time in a database by reducing the number of physical pages that a database can access.

a) Clustered Index: Clustered index is a type of index that sorts the data rows in the table on their key-value pairs. A clustered index defines the order in which data is sorted in the table which can be sorted only on one attribute from the table. In general, we use a unique attribute or a primary key to create a cluster index. Multiple Clustered Index can be created but it is not recommended due to maintenance of all Index files.

b) Un-clustered Index: Non-clustered index is a type of index which stores data at one location and indexes at another location. The index contains pointers to the locations of the data in the disk. This index improves performance when we use index only queries. There can be more than 1 non-clustered index.

3) B-Tree Index:

a) On Clustered Index: B-tree index is a balanced and shallow tree-structured index in which records are present at the last level. In this the records are sorted in the last level of index.

b) On Un Clustered Index: B-tree index is a balanced and shallow tree-structured index in which the last level of tree consists of key-value pairs. In this, the last level has a pointer which points to the respective map. B-tree is used because it needs a smaller and fixed number of searches to get any record. Shallow tree is preferred over a deep tree.

4) Batch Insert: This operation is to insert a bulk amount of data (Maps) to the database.

5) Map Insert: This operation is to insert a single map into a table in the database.

6) Joins: A join clause is used to combine data or records from two or more tables based on a common field between them. There are different types of joins namely self join, cartesian join, inner join and outer join.

7) Sort: The sort clause is used to sort the records in ascending or descending order, based on one or more columns in the given table.

8) Minibase: Minibase is a database management system that consists of implementation of different components of database management system with tests to understand the architectural flow. This enables students to understand different components of the database management system. File organization, buffer management, disk manager are few components as part of Minibase.

a) Page number: Unique id generated whenever a page is allocated.

b) Dirty bit: Page is called dirty when data is changed after reading into main memory.

c) Pin page: A page is called Pinned if the requested page is available in the buffer pool.

d) MID: Each record in a page is created with a unique ID called MID (Map ID). It is a combination of page number and slot number.

e) HF Page: Heap file page is used in heap files, it is used to maintain the page data, each record in the page is given with a map id (i.e MID)

1.2 Goal Description

The Minibase is implemented on tuples and as part of phase-2 of the project we implemented Bigtable on Maps and implemented respective methods for the execution of Minibase. Below are the few implementations from phase 2.

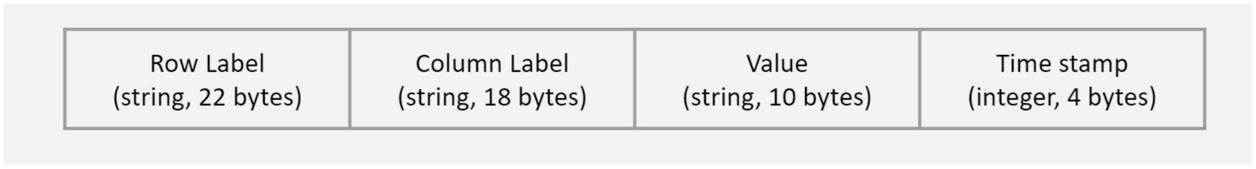
- BigT package: Creating a BigT package with three classes.
    - BigT.BigT – To create and maintain all heap files.
    - BigT.Map – To extend Minibase with Map construct and respective methods.
    - BigT.Stream – To Support getNext() feature to retrieve maps in particular order.
- Iterator.TupleUtils is modified to a new class as Iterator.MapUtils
- In diskmngr package, diskmndr.DB is modified to a new class bigDB
- Disk manager is modified in such a way that it counts the number of page reads and writes.

Phase 3 description:

- Modifying bigDB database to store data according to different storage types.
- A bigT table can now store different maps according to different storage types based on the type in which they are inserted.
- A bigT table can store at most 3 maps with the same row and column labels, but with different timestamps. This is irrespective of the batch type they were inserted and the storage type of the maps.
- A batch insert program to create an index, bulk insert maps into an existing bigT table or a new bigT table and sort the records based on the respective type.
- A map insert program to insert a single map into an existing bigT table or a new bigT table.
- A query command will access the database and return the matching maps in the requested order from a requested table with the given search conditions(equality search/ range search/ *)
- A row join program to access and row-joins two bigT tables based on the column label provided and creates a new resultant bigT table of type 1.
- A row sort operator program to sort the rows according to the most recent values for the given column name for a given bigT table and returns a new sorted bigT table.
- A get count program that returns the number of unique row labels, column labels and the total number of maps for each bigT table in the bigDB database.
- After the execution of each command, the program should also return the count of read and write pages.

1.3 Assumptions

- Map size is assumed to be 54 bytes.



Row Label (string, 22 bytes)	Column Label (string, 18 bytes)	Value (string, 10 bytes)	Time stamp (integer, 4 bytes)

- In maps we assumed the length of each record as fixed.
- Finding position of each field in the byte array since the length of each record is fixed.
- Row Label – 0 Column Label – 22 Timestamp – 40 Value – 44
- In range search, [x,y] – There is no space before and after comma null, *, [null,null] – This represents we need to consider all the values for the particular attribute [x,y] – Left value is always less than right value i.e. x<y
- Heap file page size is assumed to be 1024 bytes. Each heap file is initialized with 100k pages.
- In both the searches, Equality and Range Search the attributes need to have the first attribute in uppercase. Example – Dominica is correct, dominica is wrong declaration. This assumption was made from the data format given in test data.
- Syntax structure of a heap file is named according to storage type (1 to 5).

- Map size can be greater than a page size.
    - The data file has rowLabel length – max 9 – Format - String
    - The data file has columnLabel length – max 9 - Format - String
    - The data file has value label length – max 10 - Format – String
    - Timestamp – 4 bytes - Format – Integer. Will be given as unique value

2 PROPOSED SOLUTION

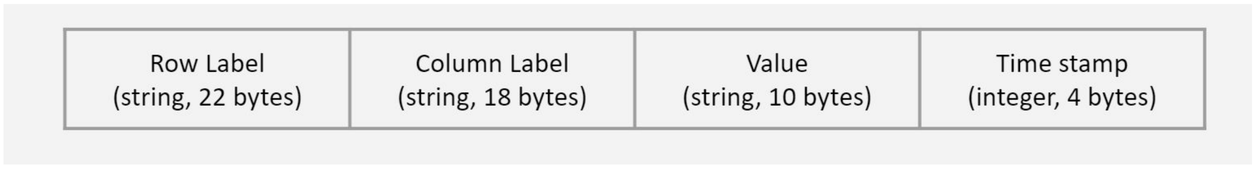
2.1 Modifications in Map data structure

2.1.1 Map Class:

We converted all the Minibase tuple implementations to the Map implementations.

The new Map construct has data stored similar to tuples but with a fixed size. The map will have four fixed attributes and we consider these attributes to be of a fixed length as it allows us to predict the storage of maps. These maps are inserted to the heap file page with a map id and this map id is a combination of page number and slot number. All these generated map ids are unique.

The byte size of the map attributes has been modified accordingly as mentioned below.



Row Label (string, 22 bytes)	Column Label (string, 18 bytes)	Value (string, 10 bytes)	Time stamp (integer, 4 bytes)

Few of the existing functions included in the map class are to initialize a new map, create a map from an existing map, copy a map, get and set row label, column label, value and timestamp.

2.1.2 MID:

MID is implemented as a global constant. This stores a map with page number and slot number. MID keeps track of each map which is spread over the Bigtable. It is used to uniquely identify a map in the Bigtable.

2.2 Modifications in project classes

2.2.1 BigDB Class:

The bigDB database can now store data according to different storage types. The modified bigDB class constructor does not take type as an input. This class extends DB.java class.

New/Modified functionalities:

- getMapCnt() - Returns the total number of maps per bigT table.
- getDistinctRowLabelCount() - Returns the total number of unique row labels per bigT table.
- getDistinctColumnLabelCount() - Return the total number of unique column labels per bigT table.
- getbigTcounts() - This method is to merge heap files of different types of a particular bigT table and return respective counts.
- runMapInsert() - This method is to insert a single map into a bigT table.
- createbigTHeapFile() - This method is to create a heap file and an index file of a respective type for a bigT table.This method is used by batch insert query. The map insertion and batch insertion is implemented in this method.
- insertMap(byte[] mapPtr) – This method inserts a new map to the db and returns its unique map ID object which is a combination of page number and slot number.
- getMap_BTreeIndex() – This method creates an index file for different storage types.
- openbigDB(String dbname) – This method is used to open the bigT database which is of the given dbname.
- closebigDB() – This method closes the current bigT database.

2.2.2 BigT Class:

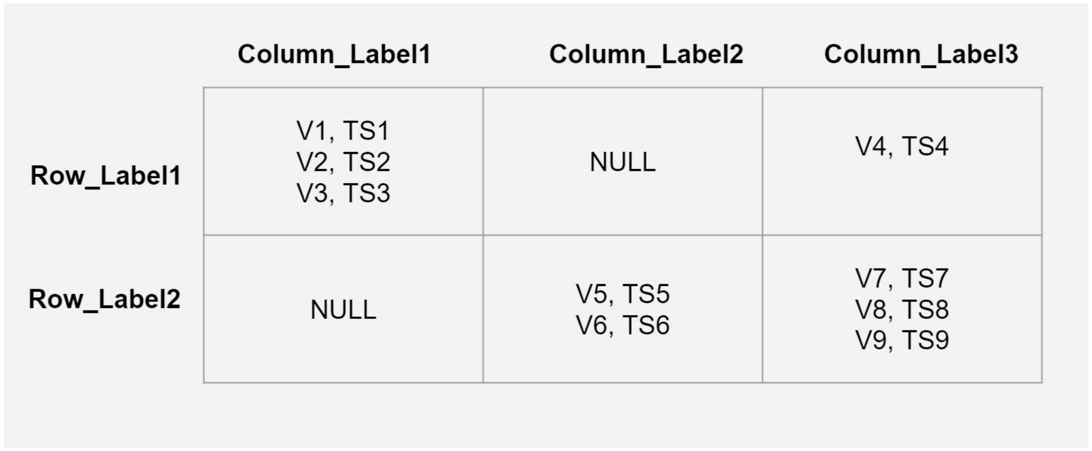
BigT class is implemented to initialize a BigT table in bigDB database.

Following the implementations from the phase 2, below are the changes that are made to the bigT in this phase.

- A bigT table can now store different maps according to different storage types based on the type in which they are inserted.
- A bigT table can store at most 3 maps with the same row and column labels, but with different timestamps. This is irrespective of the batch type they were inserted and the storage type of the maps.

For example, a bigT table can contain few maps of storage type 1, few maps of storage type 3 and few more maps of storage 5, but still it can store at most 3 maps with the same row and column labels.

Below is the sample visual representation of a bigT table in bigDB:



	Column_Label1	Column_Label2	Column_Label3
Row_Label1	V1, TS1 V2, TS2 V3, TS3	NULL	V4, TS4
Row_Label2	NULL	V5, TS5 V6, TS6	V7, TS7 V8, TS8 V9, TS9

A row of a bigT table can consist of multiple maps.

For example, (Row_Label2, Column_Label3, V7, TS7) is one map and (Row_Label2, Column_Label2 , V5, TS5) is another map but they are in the same row of the table.

Few of the existing functions included in the bigT class are to delete a bigT table, get map counts, get count of unique row labels and columns labels and insert a map.

1) bigt(String name): The type parameter from the bigt constructor is removed as to satisfy the requirements of the phase 3 . Below are the storage types:

Type 1: No index

Type 2: One btree to index row labels, maps are row label sorted (in increasing order)

Type 3: One btree to index column labels, maps are column label sorted (in increasing order)

Type 4: One btree to index column label and row label (combined key) and maps are sorted on combined key (in increasing order)

Type 5: One btree to index row label and value (combined key) and maps are sorted on combined key (in increasing order)

2.2.3 Bigtests Class:

This class is the main class of the project.

Functionalities:

- runBatchInsert() - To execute batch insert command and to insert maps in batches.
- runMapInsert() - To execute map insert command and to insert a single map.
- runSimpleNodeQuery()- To execute query command and return respective matching maps.
- runJoinQuery() - To execute a row-join command and save the resultant big table.
- runRowSort()- To execute row-sort command and sort the respective big table.
- getbigTcounts() - To execute get counts command and returns total number of maps, unique row labels and column labels.

2.2.4 Stream Class:

This class provides different types of accesses to the Bigtable:

- Stream(bigT Bigtable, int orderType, String rowFilter, String columnFilter, String valueFilter): This method takes a bigtable, order type as attributes with the filters. Based on the filters given in the query argument, the input data is filtered and sorted based upon the given order type as follows:
    - Order type 1 – the filtered results are ordered based on row label, column label and timestamp.
    - Order type 2 – the filtered results are ordered based on column label, row label and timestamp.
    - Order type 3 – the filtered results are ordered based on row label and timestamp.
    - Order type 4 – the filtered results are ordered based on column label and timestamp.
    - Order type 5 – the filtered results are ordered based on timestamp.

- closestream(): This method deletes the temp heap file created during insertions and also closes the stream close.
- getNext(MID mid): This method returns the next map in order in the respective heap file.
- ScanEntirebigT(): This method is to scan the entire bigT table.

These methods use Map ID - MID class that is declared in Global MID. Streams of maps are initialized where row label matching rowFilter, column label matching columnFilter, and value label matching valueFilter. If any of the filters are null strings, then that filter is not considered.

2.3 Modifications to existing packages

- diskmgr.PCounter.java – In this class we implemented two methods to read increment and write increment whenever a read or write operation is occurred these counters are incremented to denote the number of read and write operations needed to execute the given query.
- global.MID.java – This class has only a constructor which initializes a MID with a given page number and slot number.

- iterator.MapUtils.java – In this class we are implementing methods to compare two different maps and order the maps based upon the given order type.
- Similar to the Iterator package, a new BigTIterator and BDescIterator packages are created and modified to suit the sorting requirements of our implementations.

2.4 Programs for insertion, storage, join, sort and retrieval

2.4.1 Batch Insertion

To store maps in Bigtable we need to perform Batchinsert. Below, is the command to insert maps in batches.

" batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF "

DATAFILENAME is a string where it denotes the source of a data file.
TYPE is an integer (between 1 and 5) represents the storage type.
BIGTABLENAME is the string with which the program will store all the maps in a Bigtable
NUMBUF is an integer which initializes a specified number of buffers in the database.

The format of the input data file will be as follows:

```
row_label1  column_label1  value1  timestamp1
row_label2  column_label2  value2  timestamp2
..............
row_labelN  column_labelN  valueN  timestampN
```

When argument[0] matches batchinsert, the CSV datafile name given at argument[1] of the command line is used for storing the values in the heap file. The fields are distinguished by the comma separation between the data and are stored in their assigned and fixed fields. Each map gets a new MID on their insertion. Given these maps, the name of the Bigtable that will be created in the database will be BIGTABLENAME_TYPE. If this Bigtable already exists in the database, the tuples will be inserted into the existing Bigtable. At the end of the batch insertion process, the program will output the number of disk pages that were read and written (separately).
During the batch insert, if the given table exists in the bigDB, then we will open the existing table and continue with the map insertion. Else, we will create a new table and insert the maps according to the given storage type.

Duplicate deletion: For duplicate deletion, first we sort data based on the Row Label, Column Label and Time Stamp in ascending order. After sorting the data we are comparing the 4 consecutive maps and if the row label and column label are the same we delete the old map from the data file.

When argument[0] equals exit then the database is closed.

2.4.2 Map Insertion

Map insertion is to insert a single map into an existing table or a new table. Below is the command for the map insertion.

“ mapinsert RL CL VAL TS TYPE BIGTABLENAME NUMBUFF “

All the assumptions for batch insert will also apply here.

Implementation: First we check whether the given BigT exists or not. If it doesn't exist, we create a new bigT file for the given storage type and insert the given map into that particular bigT table. If the table is present already, we open all the respective heapfiles for the given BIGTABLE and check for duplicates and insert the map into the respective heap file.

2.4.3 Storage Types

TYPE is an integer denoting the different storage indexing strategies. Depending on the storage type (between 1 to 5), we are creating heap files and inserting maps accordingly and also generating an index file for each heap file as described below:

TYPE=1: No Index

We create a new heap file or open an existing heap file. No index file is generated with this type.

TYPE=2: One B-tree to index on row labels, maps are row label sorted (in increasing order).

TYPE=3: One B-tree to index on column labels, maps are column label sorted (in increasing order).

TYPE=4: One btree to index column label and row label (combined key) and, maps are sorted on combined key (in increasing order)

TYPE=5: One btree to index row label and value (combined key) and maps are sorted on combined key (in increasing order)

2.4.4 Query Implementation

The query command line invocation is

“ query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF “

The program will access the database and returns the matching maps in the requested order.

The attributes order of the printed maps **“ Row_label   Column_label   Timestamp   Value ”**

Each filter can be:
• “*”: meaning unspecified (need to be returned) filter, need to return all records.
• a single value – Equality Search.
• a range specified – Range Search, it is a two column separated values in square brackets (e.g. “[56, 78]”)

NUMBUF is of integer type used to define the number of buffers used to execute the query.

- In the query part, we have changed the stream in such a way that, given a bigT table name, it searches in all the available different storage type files of that particular bigT table. We make sure that it leverages the available index files of different storage type files if there are any while querying.

If ORDERTYPE is-
1, then results are first ordered on row label, then column label, then on time stamp 2, then results are first ordered on column label, then row label, then on timestamp 3, then results are first ordered on row label, then on time stamp 4, then results are first ordered on column label, then on time stamp 5, then results are ordered on time stamp.

• If TYPE = 1, full heap file scan is implemented on the Bigtable with filters on row, column and values attributes on the map. Then the given order type is implemented on the filtered maps.

The query also implements entire heap file scan when row, column and value filters are defined to be * or [null,null].

• If TYPE = 2 and row filter is not * or [null,null],scan is implemented on the row indexes. If row filter is any of null,* and [null,null] full heap scan is implemented.

Column filter and value filter can be * or [null,null]
If the column filter or value filter is * or [null,null] it returns a filtered map with indexes matching the row labels taking all column labels or value labels accordingly as mentioned in the query. - If column filter or value filter is [null,x] it returns a filtered map with indexes matching the row labels taking all column labels or value labels from the start till the value x mentioned in the query range filter respectively. - If column filter or value filter is [x,null] it returns a filtered map with indexes matching the row labels taking all column labels or value labels starting from the value x mentioned in the query range filter till the end respectively. - If column filter or value filter is [x,y] it returns a filtered map with indexes matching the row labels taking all column labels or value labels starting from the value x mentioned in the query range filter till the end where it includes the value y respectively. - If row filter is equality search and column filter and

value filter are * and range search respectively, then scan is implemented on row indexes and returns the maps that match
the row labels by taking all column labels and value labels within the specified range.

Example: query Bigtable 2 1 Singapore * [null,9300] 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists, a scan is implemented on the row indexes that is key value,MID and returns the maps of row label that matches Singapore taking all column labels and value labels from start to the value of 9300. The filtered maps are then sorted according to order type 1.

If row filter is equality search and column filter and value are also a single string which implies that it is equality search scan is implemented by searching for row labels, column labels and value labels which satisfy the given criteria.

Example: query Bigtable 2 2 Singapore Camel 9300 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists scan is implemented on the row indexes that is key value,MID and returns the maps of row label that matches Singapore and column label that matches Camel and value label that matches 9300.The filtered maps are then sorted according to order type 2.

• If TYPE=3 column filter is not * or [null,null],scan is implemented on the row indexes. If column filter is any of null, * and [null,null] full heap scan is implemented.

Row filter and value filter can be * or [null,null]

If the rowfilter or valuefilter is * or [null,null] it returns a filtered map with indexes matching the column labels taking all row labels or value labels accordingly as mentioned in the query.

If the row filter or value filter is [null,x] it returns a filtered map with indexes matching the column labels taking all row labels or value labels from the start till the value x mentioned in the query range filter respectively.

If the row filter or value filter is [x,null] it returns a filtered map with indexes matching the column labels taking all row labels or value labels starting from the value x mentioned in the query range filter till the end respectively.

If the row filter or value filter is [x,y] it returns a filtered map with indexes matching the column labels taking all row labels or value labels starting from the value x mentioned in the query range filter till the end where it includes the value y respectively. - If column filter is equality search and row filter and value filter are * and range search respectively, then scan is implemented on column indexes and returns the maps that match the column labels by taking all row labels and value label maps.

Example- query Bigtable 2 1 * Camel [null,9300] 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists, a scan is implemented on the column indexes that is key value, MID and returns the maps of column label that matches Camel taking all row labels and value labels from start to the value of 9300. The filtered maps are then sorted according to order type 1.

If column filter is equality search and row filter and value are also a single string which implies that it is equality search scan is implemented by searching for column labels, row labels and value labels which satisfy the given criteria.

Example- query Bigtable 2 2 Singapore Camel 9300 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists scan is implemented on the column indexes that is key value,MID and returns the maps of column label that matches Camel and row label that matches Singapore and value label that matches 9300.The filtered maps are then sorted according to order type 2.

• If TYPE=4, column filter and row filter are not * or [null,null],scan is implemented on the composite key of column and row indexes. If column filter and row filter are any of null,* and [null,null] full heap scan is implemented. Indices are also created on timestamp.

Row filter and column filter can be single values that runs equality search or a range that processes range search and value filter can be * or [null,null]

If value filter is * or [null,null] it returns a filtered map with indexes matching the column labels and row labels taking all value labels accordingly as mentioned in the query.

If the value filter is [null,x] it returns a filtered map with indexes matching the column labels and row labels taking all value labels from the start till the value x mentioned in the query range filter respectively.

If the value filter is [x,null] it returns a filtered map with indexes matching the column labels and row labels taking all value labels starting from the value x mentioned in the query range filter till the end respectively.

If the value filter is [x,y] it returns a filtered map with indexes matching the column labels and row labels taking all value labels starting from the value x mentioned in the query range filter till the end where it includes the value y respectively.

If column filter is equality search ,row filter is a range search and value filter is * and, then scan is implemented on the composite key of column indexes and row indexes returns the maps that match the column labels and row labels by taking all value label maps.

Example- query Bigtable 2 1 [Singapore,Maine] Camel [null,9300] 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists scan is implemented on the composite key of column indexes and row indexes that is key value,MID and returns the maps of column label that matches Camel taking all row labels within the range values and value labels from start to the value of 9300. The filtered maps are then sorted according to order type 1.

If column filter and row filter are equality search and value filter is also a single string which implies that it is equality search scan is implemented by searching for column labels, row labels and value labels which satisfy the given criteria. When the query is executed, it checks if the Bigtable is existing or not, if it exists, a scan is implemented on the composite key of column indexes and row indexes that is key value,MID and returns the maps of column label that matches Camel and row label that matches.
Singapore and value label that matches 9300.The filtered maps are then sorted according to order type 2.

• If TYPE=5, row filter and value filter are not * or [null,null],scan is implemented on the composite key of row and value indexes. If row filter and value filter are any of null,* and [null,null] full heap scan is implemented. Indices are also created on timestamp.

Row filter and value filter can be single values that runs equality search or a range that processes range search and value filter can be * or [null,null]

If column filter is * or [null,null] it returns a filtered map with indexes matching the row labels and value labels taking all column labels accordingly as mentioned in the query.

If column filter is [null,x] it returns a filtered map with indexes matching the row labels and value labels taking all column labels from the start till the value x mentioned in the query range filter respectively.

If column filter is [x,null] it returns a filtered map with indexes matching the row labels and value labels taking all column labels starting from the value x mentioned in the query range filter till the end respectively. If column filter is [x,y] it returns a filtered map with indexes matching the row labels and value labels taking all column labels starting from the value x mentioned in the query range filter till the end where it includes the value y respectively.

If value filter is equality search, row filter is a range search and column filter is * and, then scan is implemented on composite key of row indexes and column indexes returns the maps that match the row labels and value labels by taking all column label maps.

Example- query Bigtable 2 1 [Singapore,Maine] * 9300 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists scan is implemented on the composite key of row indexes and value indexes that is key value,MID and returns the maps of row label within the range values, value labels with value 9300 and all column labels which fall into the row and value filter criteria. The filtered maps are then sorted according to order type 1.

If row filter and value filter are equality search and column filter is also a single string which implies that it is equality search scan is implemented by searching for row labels, value labels and column labels which satisfy the given criteria.

Example- query Bigtable 2 2 Singapore Camel 9300 1000

When the query is executed, it checks if the Bigtable is existing or not, if it exists scan is implemented on the composite key of row indexes and value indexes that is key value,MID and returns the maps of row label that matches Singapore and value label that matches 9300 and column label that matches Camel. The filtered maps are then sorted according to order type 2.

2.4.5 Row Join

“ rowjoin BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER NUMBUF “

We have implemented sort merge join for join operation. First, we sort both the outer and inner tables using index files as the index files can be queried efficiently to fetch maps with less number of I/O’s as compared to scanning all the heap files of different storage types in a big table. Moreover, the advantage here is the seek time is only needed for the first record in the sorted index file. Here the key idea is to implement sort merge join and to achieve that we used index files for sorting purpose.

In general, in relational databases, we can compare row by row after sorting and we join two tables in a single iteration as the data is already sorted. But, here in Big DB the problem is, each row can have a chance of up to three maps and we need to consider a maps’ latest timestamp value for joining the outer and inner table. So, we came up with an idea of having only the latest timestamp maps in an index file with key as the concatenation of column, value and row (key- column_label : value : row_label) and this has been done for both outer and inner tables. As the index file can be queried efficiently, it can be used to fetch maps starting with a joined column.

In the same way as sort merge join algorithm, we can just increment the scanning over index files of both tables in the merging phase. If it was in a traditional relational database, we just retrieve the rows (each row consists of only one tuple) which were satisfied for the joined column but in big DB, the concept is like relational DB, but the physical implementation of row is different. Here in big DB, we need to retrieve rows(each row can have up to three maps) and these resulted rows have all the column labels of the two input rows except for the joined column which occurs only once in the big table and only with the most recent three values. Once we find out if any two input rows satisfies the given joined column, we just take out row labels from the two input rows and fetch all the maps from the sorted index files of two tables starting with that row labels using the query capability. For the joined column, we have used a java tree map to filter out the most recent three values.

Also, we have handled the scenario where two input rows have the same column labels other than the joined column. In this scenario, we appended the column label name with "_Right" for the right table to distinguish the records and avoid the row having more than three values with the same row and column labels. In this case, we are **not** appending "_Left" to the left table.

- Self join: A self join is a regular join, but the table is joined with itself. In this project, we can perform row-wise self join on the same bigT table with a condition on a column label.

This RowJoin class contains the following methods:

- RowJoin()- Constructor method that takes the inner table, outer table and column label as inputs. It sorts the data based upon the given column label, then compares both the tables on given condition (row wise comparison) and returns a new bigT table.
- joinBigtTables() and startJoiningTables() - This method filters maps on the latest timestamp values from both the tables with the given column filter and joins the inner and outer bigT table.

2.4.6 Row Sort

This operator results in a type 1 BigT table in which the rows are sorted (ascending or descending) to the most recent values for the given column label.

" **rowsort INBTNAME OUTBTNAME ROWORDER COLUMNNAME NUMBUF "**

Roworder = 1, Ascending

Roworder = 2, Descending

Created new packages for sort :BigTIterator and BDescIterator packages.

In this implementation, we created two temporary heap files from the input file based on column label filter. One heap file contains the maps which has the given column label and another has the maps which does not match with the given column label. We are sorting the first heapfile which has the column label maps based on the combined key of Row label and timestamp. After sorting, we are considering the latest maps from the sorted file and we are filtering the maps based on latest timestamps. The resulting maps are stored in a new temporary heapfile. Now, the result maps are sorted based on its respective value. These corresponding row labels are taken into consideration to filter the main input file and if matched, then inserted into the output file. In addition, we are considering the same filter for the first non column label filtered heap file to check if the map is present in that file. If a map exists, we ignore it as this is already present in the output file and which doesn't match with the row label that is inserted into the output file.

2.4.7 Get Counts

“ <u>getCounts NUMBUFF “</u>

We implemented getCounts functionality in the diskmgr/bigdb.java file.

This query returns the total number of maps, distinct row label and distinct column label count. For this, we merge all the respective heapfiles for different storage types of a particular bigT table and then we call getMapCnt() method of heapfile to return the total number of maps in that particular . For getting distinct row label count, first we are sorting the merged bigT file on row label, then comparing the consecutive 2 maps row label and if they are distinct we are increasing the row count by 1. After completion of scan operation the method will print the distinct count for the table. Similarly for getting distinct column labels but for this we are sorting the heapfile based on the column label. This process is repeated for all the tables present in the bigDB. We are fetching heapfile names through reading directory pages of the big DB. This method also returns the counts for rowsort and row join operation output files.

3 EXPERIMENTAL RESULTS

Given test data files: Data1.csv, Data2.csv, Data3.csv

Records from Data1 file are inserted into 5 heap files (each heap file contains 2000 records) of different storage types of a bigT table named tableA.

Data1.csv is divided into five files (Data11.csv , Data12.csv , Data13.csv, Data14.csv, Data15.csv)

Records from Data2 file are inserted into 5 heap files (each heap file contains 2000 records) of different storage types of a bigT table named tableB.

Data2.csv is divided into five files (Data21.csv , Data22.csv , Data23.csv, Data24.csv, Data25.csv)

Records from Data3 files are inserted into 2 heap files (each heap file contains 5000 records) of different storage types of the same bigT table named tableC.

Data3.csv is divided into 2 files (Data31.csv , Data32.csv)

Below are the insertions, queries and sort results of different bigT tables (tableA, tableB and tableC) with the corresponding page reads and writes.

Query	Total Maps	Page Reads	Page Writes
batchinsert C:\Users\Lokeshwar\Desktop\Data11.csv 1 tableA 200	Type-1: 1998	985	1234
batchinsert C:\Users\Lokeshwar\Desktop\Data12.csv 2 tableA 200	Type-1: 1994 Type-2: 2000	3276	3867
batchinsert C:\Users\Lokeshwar\Desktop\Data13.csv 3 tableA 200	Type-1: 1972 Type-2: 1990 Type-3: 2000	6462	7201
batchinsert C:\Users\Lokeshwar\Desktop\Data14.csv 4 tableA 200	Type-1: 1928 Type-2: 1973 Type-3: 1994 Type-4: 1999	10329	11206
batchinsert C:\Users\Lokeshwar\Desktop\Data15.csv 5 tableA 200	Type-1: 1883 Type-2: 1927 Type-3: 1972 Type-4: 1995 Type-5: 2000	13139	14105
query tableA 1 * * * 500	9777	2393	1179
query tableA 2 * * 11905 500	2	617	6
query tableA 2 * * [15393,19242] 500	399	621	10
query tableA 4 * Wren * 500	99	621	8
query tableA 4 * Wren 12216 500	1	621	8
query tableA 4 * Zebra [189,12796] 200	13	622	8
query tableA 2 * [Cirrhoscy,Sheep] * 500	7002	1020	466
query tableA 3 * [Cirrhoscy,Sheep] 39393 500	1	665	64
query tableA 5 * [Cirrhoscy,Sheep] [1235,7141] 100	396	7484	114
mapinsert Alabama Anatidae 10001 10001 1 tableA 50	Type-1: 1884 Type-2: 1926 Type-3: 1972 Type-4: 1995 Type-5: 2000	-	-
mapinsert Alaska Cobra 10002 10002 1 tableA 50	Type-1: 1885 Type-2: 1926 Type-3: 1972 Type-4: 1995 Type-5: 2000	-	-
rowsort tableA rstableA 1 Zebra 1000	9778	62119	5408

rowsort tableA rstableA 2 Zebra 1000	9778	62471	5433

Query	Total Maps	Page Reads	Page Writes
batchinsert C:\Users\Lokeshwar\Desktop\Data21.csv 2 tableB 200	Type-2: 2000	1035	1474
batchinsert C:\Users\Lokeshwar\Desktop\Data22.csv 3 tableB 200	Type-2: 1991 Type-3: 1999	3350	4037
batchinsert C:\Users\Lokeshwar\Desktop\Data23.csv 4 tableB 200	Type-2: 1961 Type-3: 1991 Type-4: 1999	6658	7654
batchinsert C:\Users\Lokeshwar\Desktop\Data24.csv 5 tableB 200	Type-2: 1917 Type-3: 1959 Type-4: 1992 Type-5: 1999	9484	10599
batchinsert C:\Users\Lokeshwar\Desktop\Data25.csv 2 tableB 200	Type-2: 3856 Type-3: 1918 Type-4: 1959 Type-5: 1995	11722	12681
query alpha2 1 CZECH_REP * * 500	0	14	2
query tableB 1 CZECH_REP * * 500	80	390	4
query tableB 2 Virginia * 65576 500	2	478	8
query tableB 2 Virginia Carcharhi * 500	3	405	9
query tableB 3 Virginia * [2013,7045] 500	7	602	10
query tableB 4 Zimbabwe Sphyrna_1 * 500	0	385	8
query tableB 4 Zimbabwe Sphyrna_l * 500	3	492	10
query tableB 1 DOMINICAN Cobra 30720 200	1	97	11
query tableB 2 Alabama Fox [2013,7045] 500	1	122	11
query tableB 5 Spain [Bullhead_,Partridge] * 200	52	713	206
query tableB 3 Turkey [Jaguar,Snipe] 97290 200	1	5921	142
query tableB 3 Turkey [Jaguar,Snipe] [20791,97290] 200	23	7264	161
rowsort tableB rstableB 2 Carcharhi 100	9278	95120	5650

Query	Total Maps	Page Reads	Page Writes
batchinsert C:\Users\Lokeshwar\Desktop\Data31.csv 3 tableC 200	Type-3: 4979	5715	6879
batchinsert C:\Users\Lokeshwar\Desktop\Data32.csv 5 tableC 200	Type-3: 1994 Type-5: 2000	13903	15362
mapinsert Alabama Donkey 10001 10001 1 tableC	Type-3: 4754 Type-5: 4978	-	-
mapinsert Canada Otter 10002 10002 1 tableC	Type-1: 1 Type-3: 4754 Type-5: 4978	-	-
query tableC 1 Alabama Donkey * 300	3	418	10
mapinsert Alabama Donkey 20001 20001 1 tableC	Type-1: 1 Type-3: 4754 Type-5: 4978	-	-
query tableC 1 Alabama Donkey * 300	3	418	10
mapinsert Alabama Donkey 50001 50001 1 tableC	Type-1: 2 Type-3: 4753 Type-5: 4978	-	-
mapinsert Canada Otter 10003 10003 4 tableC	Type-1: 1 Type-3: 4753 Type-4: 1 Type-5: 4978	-	-
mapinsert Columbia Cobra 10004 10005 4 tableC	Type-1: 1 Type-3: 4753 Type-4: 2 Type-5: 4978	-	-
query tableC 2 [Italy,Malta] * * 500	1297	627	27
query tableC 3 [Texas,Turkey] Zebra * 500	9	624	11
query tableC 1 [Serbia,Tonga] [Camel,Swan] * 200	1290	203115	598
query tableC 4 [Texas,Turkey] * [1093,3109] 200	18	48217	53
query tableC 2 [Nepal,Taiwan] [Cobra,Wren] [191,4599] 200	68	452897	376
rowsort tableC rstableC 1 Moose 200	9734	78606	5381

Counts: getCounts 50

Replacer: Clock

Number of disk pages that were read: 7122

Number of disk pages that were written: 3539

rstableA, rstableB, rstableC are respective sorted bigT tables of original tables tableA, tableB, tableC

Table Name	Total Maps	Distinct Row Labels	Distinct Column Labels
tableA	9778	100	99
tableB	9728	100	99
tableC	9734	101	99
rstableB	9728	100	99
rstableA	9778	100	99
rstableC	9734	101	99

rowJoins:

For testing purposes, we have performed row join operations on a small number of records and large number of records as well and below are the results of the row join operation on a small number of records.

Query	Total Maps	Page Reads	Page Writes
batchinsert C:\Users\Lokeshwar\Desktop\l1.csv 2 table1 200	Type-2: 7	15	11
batchinsert C:\Users\Lokeshwar\Desktop\l2.csv 4 table2	Type-4: 14	15	11

200			
batchinsert C:\Users\Lokeshwar\Desktop\l3.csv 3 table3 200	Type-3: 7	15	11
batchinsert C:\Users\Lokeshwar\Desktop\l4.csv 1 table4 200	Type-1: 5	16	11
rowjoin table1 table2 rjt12 Pinniped 20	4	123	35
rowjoin table1 table2 rjt12 Alligator 20	5	118	26
rowjoin table1 table3 rj13 Human 10	2	333	53
batchinsert C:\Users\Lokeshwar\Desktop\l2.csv 4 table2 200	Type-4: 26	27	17
batchinsert C:\Users\Lokeshwar\Desktop\l3.csv 1 table3 200	Type-1: 7 Type-3: 7	27	12
rowjoin table2 table3 rjt23 Human 50	11	23	13

(self join)

rowjoin tableC tableC selfJoinC Ostrich 150

Map count after join operation: 12922

Number of disk pages that were read: 18098

Number of disk pages that were written: 17315

Also, we have attached the output logs that contains the output of different batch inserts, queries, row-join and row-sort operations.

4 OBSERVATIONS

We have understood that sorting helps us to process data in an effective way. In this project, we have used sorting for the below tasks.

- To maintain the latest three maps of the big table.
- To join tables without looping much over the inner table.
- To obtain distinct row labels and column labels.

Also, we have understood the benefits of storing data in a sorted way on disk. Even though we have implemented an un clustered file, we were able to identify significant changes in disk read and writes as data is sorted on disk with the same key as we have in index file.

The main advantage of this storage is we are doing sequential access instead of random access while querying the table which results in a lesser number of disks reads and writes.

Also, we have observed that if we increase the number of buffers then the execution is quite faster.

5 INTERFACE SPECIFICATIONS

System Requirements: This program is best run on a UNIX machine for access to bash functionalities. Java (version 1.7 or newer) and JDK needs to be installed on the system in order to compile and run the program.

Execution Instructions: We can execute this code on any java IDE. For implementation we used Eclipse IDE with the system supporting JDK.

5 CONCLUSION

This phase of the project is the extension of phase 2 that involved the implementation of Bigtable database with basic insert and search operation. In this phase, we implemented batch insert for different storage types, a single map insert, query execution, row-join for joining two bigT tables, row-sort for sorting records in both ascending and descending order and counts module to get total number of maps, unique row labels and column labels for each bigT table in bigDB. We also implemented equality and range searches for query operations using index files. From our experimental results, we understood the importance of index files and sorting technique and the advantages of Bigtable.

6 RELATED WORK

Minibase home page - http://research.cs.wisc.edu/coral/mini_doc/minibase.html

Buffer Manager - http://research.cs.wisc.edu/coral/minibase/bufMgr/bufMgr.html
http://dbmsfortech.blogspot.com/2016/05/buffer-management.html

Disk Manager - http://research.cs.wisc.edu/coral/minibase/spaceMgr/dsm.html
http://dbmsfortech.blogspot.com/2016/05/disk-space-management.html

B+ Tree - http://research.cs.wisc.edu/coral/minibase/BTree/BTree.html

For additional understanding of the concepts below websites were used:
https://www.geeksforgeeks.org/introduction-of-b-tree/ http://dbmsfortech.blogspot.com/
https://www.w3schools.com/sql/sql_select.asp
https://www.tutorialspoint.com/dbms/dbms_indexing.htm
https://www.javatpoint.com/dbms-heap-file-organization https://en.wikipedia.org/

7 Appendix

Everyone participated in all the brainstorming sessions.
Vineeth Surya and Arun - Batch Insert, Map Insert and Get Counts
Prakash and Lokeshwar - Query and Row Joins
Kavya and Sanjana - Sort and Row Sort
Everyone contributed their part for the report and testing.