Big Table Implementation

Bhanu Preeti Anand
banand2@asu.edu

Lakshmi Sindhuja Karuparti
lkarupar@asu.edu

Sai Sruthi Mallineni
smallin1@asu.edu

Sai Uttej
sthunugu@asu.edu

Shantanu Aggarwal
saggar24@asu.edu

Srija Saha
ssaha35@asu.edu

April 23, 2020

Abstract

This project aims at building strong foundation and intricacies of Database Management System and Implementation. Minibase (a database management system that is intended for educational use) is considered as a base for this phase and changes relevant to Big Table are made on the top of it. The scope of the third phase of the project is to add some advanced functionalities to the second phase by introducing join and sort operations.This focuses on implementing a different version of batch insert,map insert, query execution,row sort and row join.For the insert functionalities, we would implement different storage mechanisms (heap files, secondary indexes based on B+ Trees).
Keywords: *Big Table, Maps, B+ Trees, Buffer Manager, Heap Files, Sort,Sort merge join, Conditional Expressions, Stream,Row sort,Row join*

1 Introduction

Big table is a compressed and high performance data storage system built on google File systems[1]. The data is stored in the form of key-value pairs along with time stamps to achieve versioning. In Phase 2, minibase modules were used as building blocks to create a big-table like database, in which maps were inserted and queried in a single big table.

But in this phase, some advanced operations like grouping of similar maps, rowjoins, count and rowsort have been achieved. A database comprises of more than one big table. In each big table, maps are stored in five different storage types as per the requirement.

Various statistics within the bigt can be achieved with the get counts functionality. Joins are one of the most costly operations in DBMS and in this phase, we perform row join on multiple big tables leveraging sort-merge join technique. While a sort operation will sort the rows based on order type and a coulmn name.

In addition to batch insertion, single map insertion can also be performed. The Query is performed on the entire big table and is not restricted to a particular type which is efficiently performed using indexing. With these above enhancements, the minibase relational database has been used to construct a powerful bigtable on it.

1.1 Terminology

- Type: type refers to a storage type inside a bigt. 1 refers to no index, 2 refers to row-sorted maps and an index file indexed on row, 3 refers to column-sorted maps and

an index file indexed on column. 4 refers to column+row sorted maps and an index file indexed on column+row. 5 refers to row+value sorted maps and an index file indexed on row+value.

- Batchinsert: Insertion of a set of maps into a big table based on the type.
- Mapinsert: Insertion of a single map into bigt based on type
- Versioning: Maintaining utmost 3 maps with the latest timestamps by removing duplicates
- Query: Returns a stream of maps from a big table in a specific order
- Rowjoin: Joining two rows in a bigtable based on the column name
- Rowsort: Sorting the rows of a bigtable in a specific sort order based on a column name
- Conditional Expression: The condition which is used in retrieving specific maps based on row and column label filters.
- Sort-merge join: It is join algorithm in which the tables are sorted based on join attributed and simultaneously scanned.
- Bigtable/Bigt:It comprises 5 heap files of types 1-5 and 4 index files of types 2-5.

1.2 Goal Description

The goal of this phase is to build on the previous phase by introducing efficient storage techniques and other advances operations like join and sort.

- getCounts: The count operation will provide statistics of maps, distinct row labels and distinct column labels of every bigtable in the database using utmost n buffers.
- To modify storage strategies of phase-2 by constructing a single big table for five different storage types. In type1, maps will be stored randomly. While in type 2 and 3, maps will be sorted and stored based on row and column labels respectively. In types 4 and 5 , maps will be sorted based on the combined key used for indexes.
- Each BigDB corresponds to a database which in turn contains a Big table with an indexing strategy. A set of maps will be inserted via batch insert.A single map will be inserted via map insert.
- Query processing will fetch multiple maps from a big table irrespective of the types using utmost n buffers.
- Join is one of the most important operations in a database.A row join will be implemented to join rows for a given column names using utmost n buffers.

- Rowsort will be implemented to sort rows on the most recent values of a given column name based on two different row orders (ascending and descending).
- Number of disk accesses for reads and writes will be tracked for every operation, be it insertion of records or their retrieval.

1.3 Assumptions

- Size of Map is fixed ie.,82 Bytes which also included the header field that contains metadata about the map
- Size of the row label and column labels must not exceed 20 characters after row join is performed
- We are dealing with unclustured b+tree indexes only.
- Value of the timestamp must be in the range of [-2147483648, 2147483647].
- We proceed with an assumption of timestamps being inserted in an ascending order.
- A minimum of 150 buffers are needed to perform rowjoin, rowsort and getcounts operations.
- The column filter must be present always in the bigt that is given as input for sorting and join.
- The time mentioned in all the tables under Results is in seconds.

2 Description of the proposed solution/Implementation

2.1 Architecture

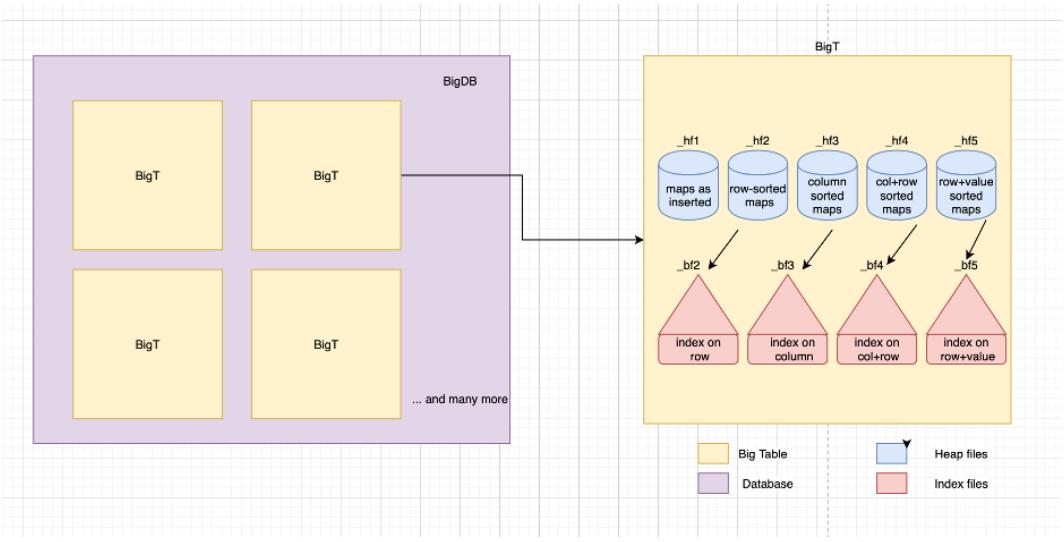

Figure 1: Architecture

A single DB consists of multiple Big Tables. Each Big Table comprises of 5 heap files, one for each type and 4 index files for types 2-5. All these are initiated every time a bigt is created.

2.2 BatchInsert

Command for Batch Insert
BatchInsert filePath TYPE BIGTNAME BUFFERS

Batch insert is performed for insertion of maps into different heap files of bigt based on the type. For types 2,3,4 and 5, it is required to maintain sorted order in the respective heap file based on the key specific to each of it. To achieve the same, a temporary heap file is created with the existing maps from heap file of Big table. This follows the deletion of the existing heap file, and the data in temporary file is now sorted. Then the maps are inserted into the corresponding heap file based on the type specified by the user . Now that records in the heap file are sorted, temporary heap file is deleted. Versioning is then performed on all the maps in bigt with the help of an utility index file. However, for type 1 since sorted order need not be maintained, maps are directly inserted into the heap file and versioning is performed, temporary heap file is not needed in this case. Index files are created for types 2-5 every time a bigt is initiated along with the heap files.

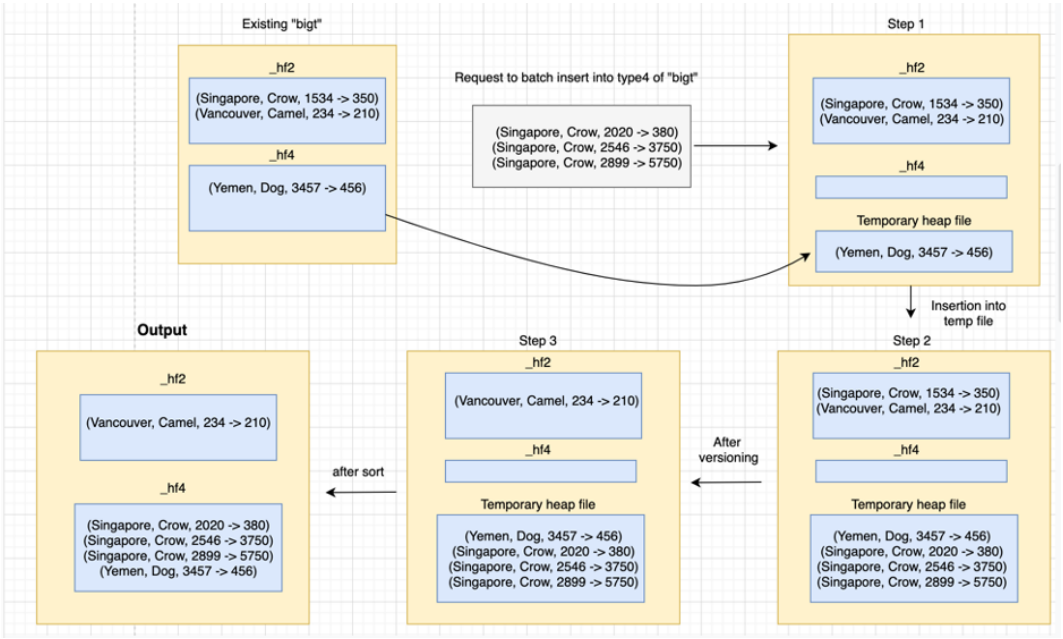

Figure 2: Batch Insert

2.3 Map Insert

Command for Map Insert
mapInsert RL CL VAL TS TYPE BIGTABLENAME BUFFERS

A map is constructed from the given rowLabel, ColumnLabel, timestamp and value and is inserted into the bigtable with the given bigtname. If the bigTable does not exist a new bigtable with the given name is created. If the given type is 1 the map is inserted into the heap file corresponding to type1. For other types since the records are already in sorted order, the correct position of the input map is found and is inserted at that position in the following way:

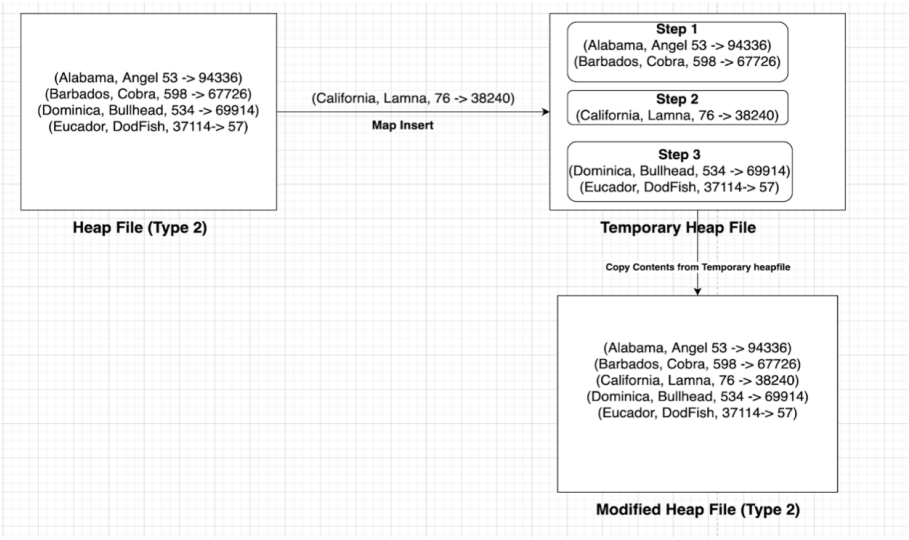

Figure 3: Map Insert

A temporary heap file is created to copy the maps from the given heap file which already in a sorted order. A scan is opened on the heap file and according to its storage type all the maps that come before the given input map are inserted into the temporary heap file. Now the input map is inserted into its correct position in the temporary heap file. After this step the scan is continued on the actual heap file and all the records that occur after the input map are copied into the temporary heap file. Now all the maps including the given input map for map insert are inserted in the sorted order in the temporary heap file. A scan is opened on the temporary heap file and all the maps are copied in the same order into the original heap file. Thereby a map insert operation is performed by performing two sequential scans, without sorting the contents of the entire heap file.

Versioning in Map Insert If there are more than three maps with the same row label and column label of the given input map in the entire bigt, the map with the earliest time stamp has to be deleted. After inserting the given input map into its appropriate storage type, a query operation is performed on the entire bigt by setting a conditional expression with the given rowLabel and columnLabel. If the result of the query returns less than or equal

to three maps then deletion is not required. If the query returns more than 3 maps with the given row label and column label, the timestamps of all the maps are compared and the map with the least timestamp is deleted from the bigt.

2.4 RowJoin

Command for Row Join
RowJoin INBTNAME1 INBTNAME2 OUTBTNAME COLUMN BUFFERS

Row join has been implemented in order to join two rows from separate big tables with certain predefined conditions of equality join that bears a resemblance to a relational database. Two rows can only be joined if they have a same column label with similar value for the latest time-stamp. Join has been implemented with the help of Sort-Merge join which will sort a stream of map on the basis of conditional attributes for join. This will reduce the computation required to traverse the inner big table. Following to this, two separate streams are maintained in sorted order and iterated such that run-time complexity is not more than O(m+n+mlogm+nlogn) for this operation. If we find a pair of rows that satisfies the join condition, an evaluation function will be called in order to join both of the row labels with the help of all the columns that lies under a particular row label. Due to design constraints, we have kept only latest three values for the column mentioned in the join condition while we are keeping all the other columns under given rows. Architecture for the row join is as follows:

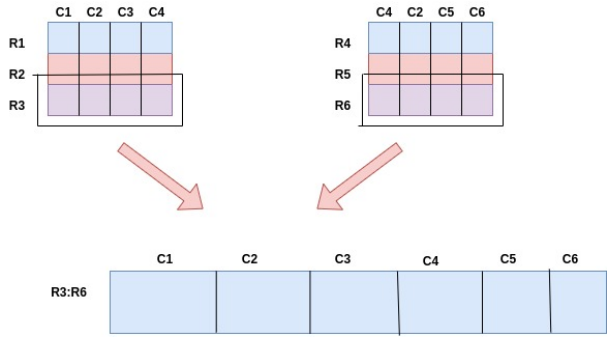

Figure 4: Row join

Two rows can only be joined if they have a common column and the values for the latest timestamps are equal for both of these rows. We have implemented some classes and functions in order to achieve the functionality.

2.5 Row Sort

Command for Row Sort
rowSort INBTNAME OUTBTNAME ROWORDER COLUMNNAME BUFFERS

Given an input Bigtable, a row sort operator is used to sort the rows based on the most

recents values for a given column name. It is supported by the getNext() function, which returns the resulting maps of the output bigtable after performing the row sort in a specified order(ascending or descending). All the resultant maps of the rowsort are added into storage type1 of the output bigtable.
Initially in a row sort operation, all the maps with the matching column name are sorted based on row label and timestamp. From these resultant maps, which are filtered based on the given column name, all distinct rows with their most recent values based on the timestamp are stored in a temporary heap file. Since rowSort is based on the values of the row labels, the maps are sorted based on the values and stored in the temporary heap file. After the sort operation is performed, each row label is retrieved and all the maps with this matching row label are obtained by performing a query on input bigt and stored in output bigt. All the rows that do not have matching column names are also stored in the output bigt.

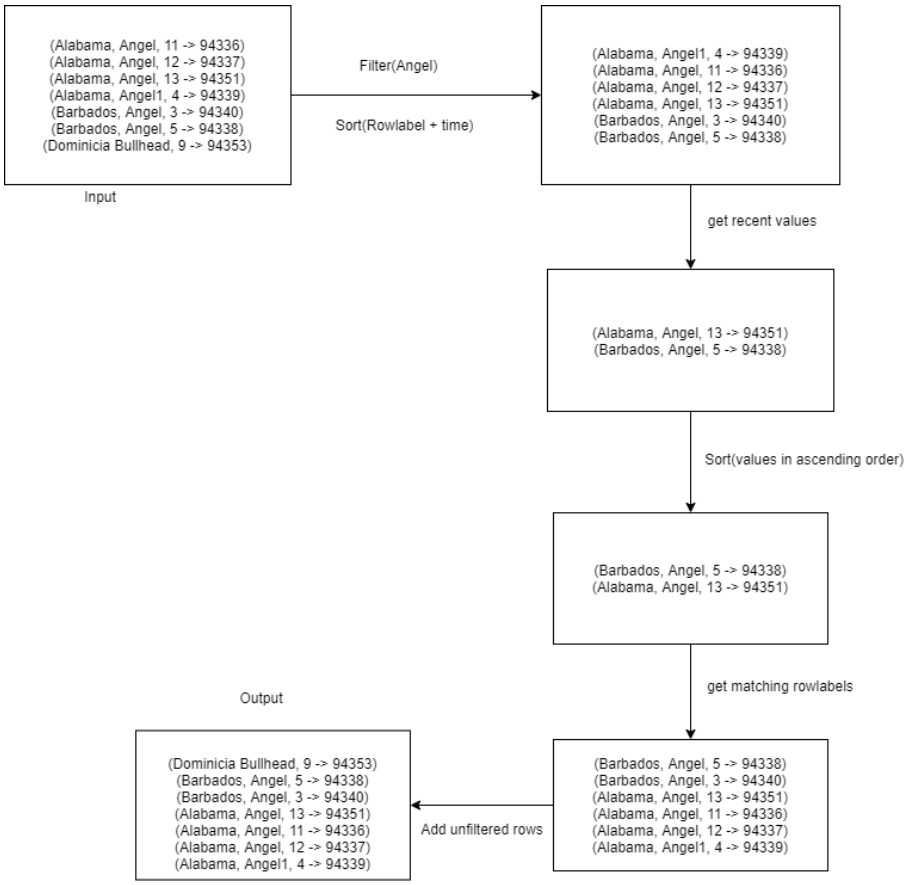

Figure 5: Row Sort

The figure shows Rowsort operation on column name 'Angel'

.

2.6 Get Counts

Command for Get Counts

getCounts BUFFERS

Given a bigTable, this gives the total number of maps, distinct row labels, column labels. It initializes a multiple file scan on the bigt and then calls the getMapCnt(), summing them to get the total number of maps in the bigt. To get the count of distinct row labels, a stream is initialized on the bigt and the resultant maps are sorted according to the row labels. Given the sorted order, the row labels are compared to get the distinct count within the bigt. Column count is also calculated in the similar fashion.

2.7 Query

Command for Query
Query INBTNAME TYPE ROWFILTER COLUMNFILTER VALUEFILTER BUFFERS

Query is initialized using a stream of maps depending on the filters that the user provides. As there is no specific file, the query has to access all the heap files in the bigt. As each file uses different kinds of indexing, a single scan does not suffice the requirement.
Consider querying on a heap file with index on row label, a filter on row with either range or single value can be achieved efficiently using an index scan. For other filters, filescan seems to be more efficient.
A new class QueryScan is used for this purpose wherein the eval expression and index expression are initialized as per the filter values. This expression is initialized considering if the filter is a independent value, range of values or a "*". In case of an independent value or range of values, the expression is constructed such that the index file is scanned to get the specific values. This approach for range of values changes in case the index file is built on a combined key and hence the expression takes in null. File scan is initialized on the first file as it has no indexing and the index scan with appropriate filters are initialized on the other files.

Once the stream of maps is returned, the maps are sorted as per the users requirement using Sort that uses the comparator within MapUtils Class.

3 Interface specification

3.1 Batch Insert

- **void insert(BigT b, String dataFile, int numbuf, int type)**

  In this method, data is read from dataFile line by line, converted into a map and inserted into specific heap file in BigT b based on the type. This is done using numbuf number of buffers. *createTempHF* and *maintainSortedOrder* are defined to make sure maps are inserted in sorted order based on type. *populateBtree* and *removeDuplicates*

are used to enable versioning in the entire BigT b. *insertIndex* is used to generate indices

- **MID insertMap(byte[] mapPtr, int type)**

  This function inserts byte array of the map, mapPtr into heap file of type 1 (*_hf1*) inside BigT b if type is 1 else it inserts into temporary heap file. It returns mid of the inserted map as the output.

- **void createTempHF(int type)**

  If type is 2,3,4 or 5, existing maps from respective heap files (*_hf2* or *_hf3* or *_hf4* or *_hf5*) are inserted into a temporary heap file, once this is done, respective heap files are deleted and new ones with the same names are initialized again so that the sorted maps would be inserted later.

- **void maintainSortedOrder(int type, int numbuf)**

  FileScan is initiated on temporary heap file and is passed as an argument to Sort constructor. Through get_next functionality of Sort.java, maps are retrieved one-by-one in the sorted order on keys based on the type and then inserted into respective heap files(*_hf2* or *_hf3* or *_hf4* or *_hf5*). Now that sorted records are inserted into respective heap files, purpose of temporary heap file is met and so, is deleted at the end.

- **void populateBtree()**

  This function acts as a first step for versioning. It creates a temporary utility index file, BTreeFile with key on row label+column label and timestamp. It is required to perform versioning on all maps inside bigt in this phase. Hence, file scan is done for all the maps inside bigt. Map along with it's mid is retrieved using get_nextMidPair defined in MutipleFileScan class, key is formed as above and both mid and key are inserted into index file. This continues until indices for all maps in the entire bigt are inserted into index file.

- **void removeDuplicates()**

  This function is the second and final step for versioning. After BTreeFile is created in the first step, Index Scan is initiated on this. If row label+column label matches, they are put into a list, if the list size is 4, the first element in this list is removed since it has the least timestamp(BTreeFile makes sure timestamps are ordered). The mid of the map that is to be deleted can be from any of the 5 different types of heap files in bigt. *deleteRecord* from Heapfile.java is used for this deletion.

- **void insertIndex(int type)**

  Based on the type, index file is created by scanning the heap file and generating the indices on the type specific key. When a bigt is initiated, all the index files of types 2-5 are created.

- **MapMidPair get_nextMidPair()**

  This is a function defined to get both the map(all fields) and it's corresponding mid when Scan is initiated using MultipleFileScan. This scan return records of entire bigt.

3.2 Map Insert

- **public void mapInsertOrder(int type, Map mapInsert)**
  This method takes an input map and the storage type for inserting it as input parameters and inserts the map into its corresponding storage type. A scan object is opened on the heap file of the given type and the records are inserted into a temporary heap file until the insert position for the given map is obtained. The input map is inserted into its correct position in the temporary heap file and the scanning of the input maps and insertion into the temporary heap file is continued until the end of the heap file. Now the actual heap file and index file are destroyed and recreated. A scan object is created on temporary heap file and the records are inserted in sorted order into the original heap file and index keys are also inserted into the corresponding Btree file.

- **public void removeDuplicatesMapInsert(String rowLabel, String colLabel)**
  This method removes duplicates after performing a mapInsert operation. The row label and column label of the mapInsert operation are taken as input and all the heap files present in the bigt are scanned for matching rowlables and columns labels by setting the conditional expressions. If the map count of the result is more than 3 then the MID of the map with the least time stamp is retrieved. Based on the MID the heap file corresponding to this record is identified and is removed from the heap file and the corresponding Btree file if it exists.

3.3 Row Join

- **RowJoin(int mem,Stream leftstream,String rightBigtable,String Columnfilter,String output)** : Rowjoin class is implemented in addition to the BigT class which takes the amount of buffers that must be initialized in order to implement the row join. In addition to this, we also specify the left stream of maps with an ordertype between 1 and 5 and these rows will be joined with rightBigTable based on a specified column filter. Resulting rows will be stored in a separate bigtable of type 1.

- **CreateHFRows(Stream s,Heapfile hf)**: The major job of the given function is to initiate a stream on inner and outer table and fetch the latest values based on the timestamp for every unique row present in that particular stream based on column filters.All these records are inserted in a separate heap file to avoid over-allocation of buffers.

- **SortMerge():** SortMerge class has been inherited from minibase functionality in order to implement sort-merge join on two bigtables. This would reduce the cost of

looping sequential scan on the inner bigtable. It takes two seperate streams as an input and tries to sort it with respect to the values that would mitigate the complexity of joins.

- **getnext()**: This function implements an iterator over the results of sortmerge in order to join two rows based on the given conditions.A pool of three buffer pages are utilized for join evaluation and result is written in an output buffer.

- **constructbigt()**: Following function would come into play after the getnext() operator has provided us with all the row pairs that can be joined. We will open a stream in order to fetch each and every column under both of these rows and append results to an output bigtable with an indexing of type 1. Join algorithm has been optimized in such a way that total complexity won't exceed O(M+N+MlogM+NlogN).

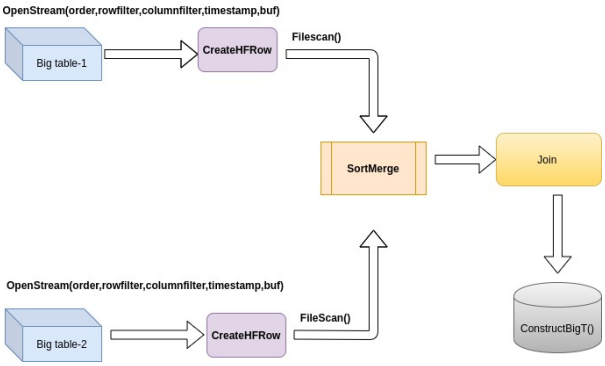

Figure 6: Row join architecture

3.4 Row Sort

- **public void performRowSort(BigT ib, BigT ob, String columnName,String orderType, int numBuf)**
  This is the major function that performs rowSort and stores the resultant maps in the output big table according to the specified sorted order. Initially all rows that do not have matching column names are stored in the output bigtable.
  Then all the distinct rows with the matching column labels are stored in a heap file based on their most recent timestamp. The maps present in this heap file are sorted based on the values using Sort() operator provided by minibase. Maps are retrieved from the sort operator using sort.getnext() and a query operation is performed to retrieve all the maps with the matching rowLabel by constructing a conditional expression with the given rowLabel passed as input to the query and the resultant maps are stored in the output big table.

- **getNext()**
  It returns all the maps in a specified order by performing a file scan on output BigTable

- **public void initEval(String rowLabel)**
  Initializes a conditional expression with the given row label to retrieve all the maps with this matching row label.

- **public void copyUnFilteredRows(Heapfile hfRow, Heapfile hfTemp,BigT ib, BigT ob)**
  Copies all the rows without the matching column names into output bigtable. This operation is performed by comparing the two heap files passed as input. Heapfile hfRow consists of all the row labels with their most recent values and heapfile hfTemp has the rows filtered based on the given column name. All the rows that are present in hfRow but not in hfTemp are retrieved and a query operation is performed to get all the maps with these matching row labels and are stored into output bigt.

3.5 Get Counts

- **int getMapCnt()** This returns the total number of maps in the bigt.

- **int getRowCnt(int numbuf)** This function initializes s a stream which in turn performs a query scan and then sort the maps according to rowlabel. This stream of maps can be used to compare and calculate the distinct rows in bigt.

- **int getColumnCnt(int numbuf)** This function initializes s a stream which in turn performs a query scan and then sort the maps according to Column label. This stream of maps can be used to compare and calculate the distinct columns in bigt.

3.6 Query

- **Map get_next()**
  This function is used to retrieve maps from the stream initialized by the constructor. This iteratively returns the tuples from filescan followed by the index scans of the corresponding heap files within a bigt. This inturn calls sort.get_next() that returns maps in increasing order.

- **Void close()**
  This function closes the appropriate files and scans that have been used in streams.

- **QueryScan(String file_name, AttrType in1[], short s1_sizes[], short len_in1, int n_out_flds, FldSpec[] proj_list, CondExpr[] outFilter, CondExpr[] indExpr, CondExpr[] indExpr2, CondExpr[] indExpr3, CondExpr[] indExpr4)**
  This function returns the maps in the bigt with the specific filter expression. The maps in the first file are accessed using a filescan, the scan on the second file is a index scan which is efficient in case of an expression with row label. An index scan is initialized with the eval expression and the corresponding index expression. Similarly, index scan is performed on the other files.

3.7 Helper functions

- **void scanBigT()** This is a helper function created in order to scan each heap file in bigt and print it. Scan is initiated using openScan function from Heapfile.java and records are retrieved using *getNext* functionality in it.

- **Void compareMapWithMapForSorting()**
  This function in MapUtils is used as a comparator that sorts the data depending on the given order. Various cases define on the users requirement on how to sort the data.
  Various cases that sort the data as per the required order to insert data into bigT is defined in the cases 7 to 10. The other cases are used for the purpose of sorting.
  As an example, Case 1 is used to sort the data initially based on RowLabel, followed by Column label and Timestamp. Similar pattern is observed in other cases depending on the attribute type being either String or an Integer.

4 System requirements/installation and execution instructions

- Install Linux system with an appropriate linux distribution such as Ubuntu.
- Download and extract the cse510_bigtable file.
- Modify the ”Makefile”s to reflect your directory structure.
- In the ”src” directory, build the source by running ”make db”.
- Run the mainClass.java in tests folder.
- When the interface prompts you to enter, choose the option of either batch insert,map insert, query, getCounts, Row Join, Row Sort based on the requirement.
- On choosing batch insert, it prompts to enter the query wherein the query is inserted with the following parameters.
  1. batchInsert (type of operation)
  2. filepath (path of the file)
  3. type (type of db and indexing schemes)
  4. BigT Name (name of the bigt)
  5. numbuf (number of buffers)

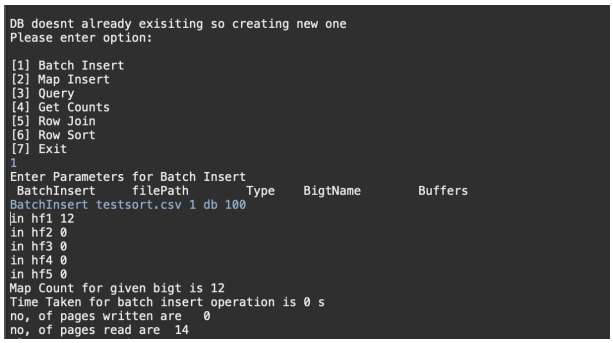

Figure 7: Batch Insert

- On choosing map insert, it prompts to enter the query wherein the query is inserted with the following parameters.
  1. mapInsert (type of operation)
  2. Row (Row Label)
  3. Column (Column Label)
  4. Value (Value)
  5. TimeStamp (TimeStamp)
  3. type (type of db and indexing schemes)
  4. BigT Name (name of the bigt)
  5. numbuf (number of buffers)

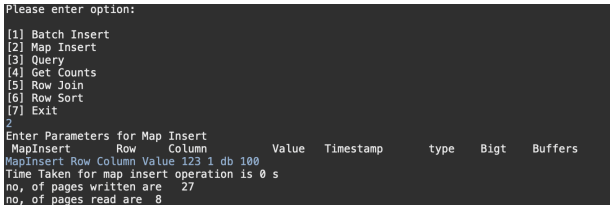

Figure 8: Map Insert

- On choosing RowJoin, enter the query operation with the following parameters.
  1. query (type of operation)
  2. BigTName1 (name of the bigt)
  3. BigTName2 (name of bigt)
  4. outputBigt (name of bigt)
  5. Column (Column to perform Join On)
  6. numbuf (number of buffers)

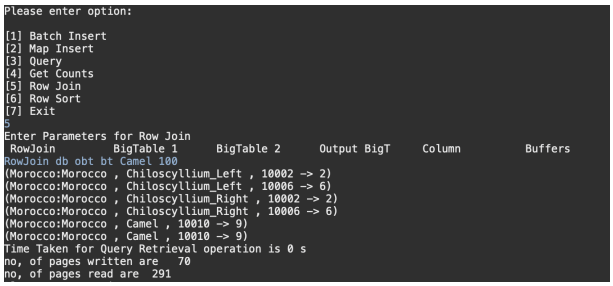

Figure 9: Row Join

- On choosing RowSort, enter the query operation with the following parameters.
  1. query (type of operation)
  2. BigTName1 (name of the bigt)
  3. outputBigt (name of bigt)
  4. Column (Column to perform Join On)
  5. Order (Order to sort)
  6. numbuf (number of buffers)

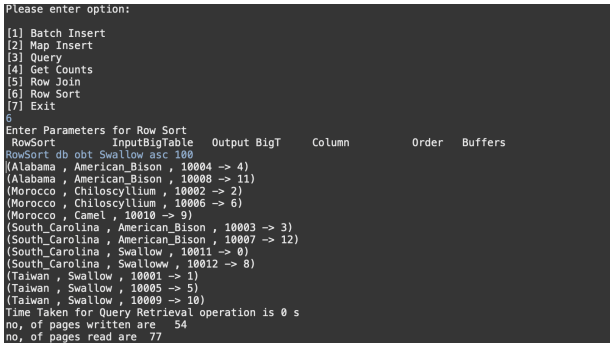

Figure 10: Row Sort

- On choosing getCounts, enter the query operation with the following parameters.
  1. getCounts (type of operation)
  2. numbuf (number of buffers)

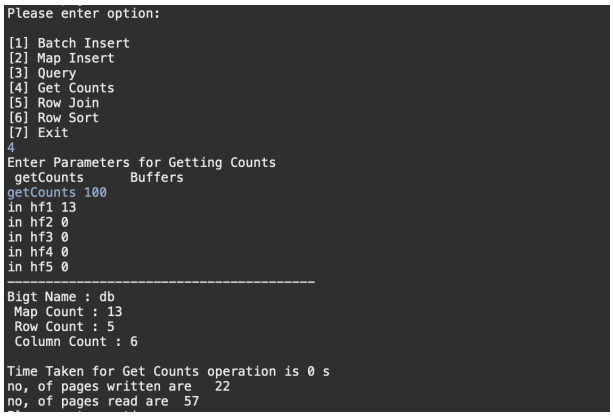

Figure 11: Get Counts

- On choosing query, enter the query operation with the following parameters.
  1. query (type of operation)
  2. BigTName (name of the bigt)
  3. ordertype (sort order of the results)
  4. rowfilter (filter on the row either *(no filter), range of values inserted in square braces, individual value)
  5. colFilter (filter on the column)
  6. valueFilter (filter on the value)
  7. numbuf (number of buffers)

Figure 12: Query

5 Results

5.1 Batch Insert

dataSize	BigtType	Buffers	Read	Write	Time
10000	1	200	6228	6235	3
10000	2	200	12794	11718	8
10000	3	200	20157	16794	9

Figure 13: Batch Insert

Three batch insert operations are performed consecutively on 3 different storage types (1,2 and 3) of a single bigtable and the page reads reads and writes are depicted in the above

table. As we can see as the number of records are increased from 10,000 to 20,000 during the second batch insert operation the reads and writes are also doubled since the number of buffers provided are still the same. The time taken to perform the batch insert operation is also increased in each step.

Batch Insert with different amount of buffers

dataSize	BigtType	Buffers	Read	Write	Time
10000	1	100	9818	9095	3
10000	2	100	16407	14709	7
10000	3	100	31796	23504	9
dataSize	BigtType	Buffers	Read	Write	Time
10000	1	200	6228	6235	3
10000	2	200	12794	11718	8
10000	3	200	20157	16794	9
dataSize	BigtType	Buffers	Read	Write	Time
10000	1	300	4472	4498	3
10000	2	300	11003	9921	9
10000	3	300	16387	13701	12

Figure 14: Batch Insert with Varying Buffers

This table depicts the page reads and writes along with time taken for batch insert operations of the same data when the buffers are increased by 100 at each step.

5.2 Map Insert

Map Insert vs Batch Insert

dataSize	BigtType	Buffers	Read	Write	Time	totalRecords
10000	2	200	4983	4585	7	9777
1	2	200	3433	2916	6	9778
10000	2	200	12311	9958	31	17727
1	2	200	6188	5062	20	17728
10000	2	200	18466	14162	53	23040
1	2	200	8168	6510	31	23041

Figure 15: Map Insert vs Batch Insert

Map Insert operation is performed on a bigt after doing a batch insert operation and the above table depicts the change in the page counts and time taken between the two for the same number of buffers provided as input.

1) First, 10,000 records from test data file Data1.csv are inserted into bigt with storage type2. Read count for this operation is 4893, write count is 4585 and the time taken for this operation is 7 seconds with 200 buffers provided as input. Then a map insert operation is performed on the same bigt with the same storage type and read count and write counts are 3433 and 29166 respectively with 200 buffers provided as input.

2)Then another 10,000 records from test data file Data2.csv is inserted into the same bigt and storage type with the 200 buffers. The read counts and write counts are 12311 and 9958 respectively. Now a map insert is performed on the table with same number of buffers and the page counts are only 6188 and 5062 for reads and writes.

3) Similarly the last set of 10,000 records from the test data file Data3.csv are inserted and a map insert operation is performed with same input conditions.

As we can see, since we are not sorting the entire data present in the heap file for a map insert operation as the number of records in the bigtable increases the page counts and time taken is drastically reduced. In the above table when there are 17000 records in the db a batch insert for 10000 records takes 53 seconds and page counts for read and write are 18466 and 14162 respectively whereas the map insert only takes 31 seconds and page counts are also reduced by more than half when compared to batch insert.

5.3 Row Join

datasize	buffers	read	write	time
30000	150	1913876	80466	68
30000	200	1819336	68825	67
30000	250	1666804	56639	68

Figure 16: Row Join

The rowjoin operation is performed on a data file with 30,000 records and varying number of buffers. The above table shows the time taken, number of reads and writes to perform the rowjoin operation.
The time taken to perform a rowjoin on 30,000 records with 150 buffers is 68 seconds with a read and write count of 1913876 and 80466 respectively. Similarly when the number of buffers are increased to 200, then the time taken is 67 seconds with a read and write count of 1819336 and 68825 respectively. For 300 buffers, the read and write count are 1666804 and 56639 in 68 seconds.
Basically rowjoin involves opening of two streams for left and right tables followed by a filescan on a temporary heap file. Additionally sort operations are performed in the phase of sort-merge join. These steps lead to more read and write counts. It is observed that in spite of increase in number of buffers, the time taken is the same.

5.4 Row Sort

datasize	buffers	read	write	time
30000	150	652223	34639	35
30000	200	522234	23106	31
30000	250	416262	15049	30

Figure 17: Row Sort

The rowsort operation is performed on a data file with 30,000 records and varying number of buffers. The above table shows the time taken, number of reads and writes to perform the rowsort operation in ascending order.
The time taken to perform a rowsort on 30,000 records with 150 buffers is 35 seconds with

a read and write count of 652223 and 34639 respectively. Similarly when the number of buffers are increased to 200, then the time taken is 31 seconds with a read and write count of 522234 and 23106 respectively. For 300 buffers, the read and write count are 416262 and 15049 in 30 seconds.
Basically rowsort includes stream, multiple file scans, creation of heap files and sort operations, which justifies the number of reads and writes. It can also be observed that the time taken, read and writes have decreased due to increase in the number of buffers.

5.5 Query

RowFilter	ColFilter	ValFilter	Buffers	Read	Write	Time	Test data Size
*	*	*	200	5945	3108	12	30000
Alabama	*	*	200	2306	418	2	30000
*	Camel	*	200	2238	360	2	30000
*	*	14692	200	3552	779	4	30000
Alabama	Camel	99295	200	992	44	0	30000
Alabama,Morocco	*	*	200	4196	1750	4	30000
*	Camel,Fox	*	200	2963	849	3	30000
*	*	[20000,30000]	200	3781	966	5	30000
Alabama,Morocco	Camel,Sheep	[20000,30000]	200	264	1891	1	30000

Figure 18: Query using Index Scan

We are using indexes created during batch insert and map insert for efficient retrieval of maps during query operation.Three batch insert operations are performed consecutively on 3 storage types (1,2 and 3) of same bigt using test data files Data1.csv, Data2.csv and Data3.csv with each file consisting of 10k records. All possible combinations of row, column and value filters are executed and page counts and time taken for each of these query operations are depicted in the above picture.

Query using FileScan vs Index Scan

For efficient retrieval of records during query operation we are using indexes that are created on the storage types. The following table depicts the differences in the page count and time taken for a query operation using file scan and index scan.

FileScan					
Row Filter	Col Filter	Val filter	read	write	time
Alabama	*	*	1864	204	0
*	Camel	*	1864	204	0

Index Scan					
Row Filter	Col Filter	Val filter	read	write	time
Alabama	*	*	60	209	0
*	Camel	*	63	209	0

Figure 19: Query using File Scan vs Index Scan

Batch insert operation is performed on a bigtable with storage type 2 and 10k records are inserted into it. When a query operation is performed using the file scan (which traverses the entire bigt to find matching records) the read count is 1864. When a same query operation is performed using index scan(which traverses the index tree built on the row

labels to retrieve the matching records) the read count is only 60.

In the similar way a batch insert operation is performed on a bigtable with storage type 3 and 10k records are inserted into it.When a query operation is performed using the file scan (which traverses the entire bigt to find matching records) the read count is 1864. When a same query operation is performed using index scan(which traverses the index tree built on the column labels to retrieve the matching records) the read count is only 63.

Therefore the indexes are efficiently for query operations.

5.6 Get Counts

Database	buffers	read	write	time
db	200	11913	6087	13
db2	200	23861	11972	27

Figure 20: get Counts

The count operation is performed on a data file with 30,000 records with utmost 200 buffers. The above table depicts the time taken, number of reads and writes to perform the getCounts program when one and two big tables exist in a database.
Initially, the number of maps, distinct row labels and distinct column labels in the database, only when one big table existed, are 23038, 100 and 99 respectively. The time taken is 13s, while the read and write counts are 11913 and 6087 respectively. Because a stream is initialized with the big table, which internally file or index scans and sorts the data which suffices the read and write counts.
When two bigtable exist, the time taken is 27s, read and write counts are 23861 and 11972 respectively. It is more when compared to one bigtable because a stream is initialized on every bigtable present in the database. Thereby justifying the increase in time, write and read counts.

6 Related work

Initially ,big table was developed by google in order to design a distributed storage system that could scale well.Motivation behind big table was to provide a simple data model that could provide dynamic control over data layout.Google devised this data model such that:

- Big table consist of a row family where all the row keys are stored in a lexicographical order and they are dynamically partitioned into tablets.
- Column keys are also grouped into a column family but a column key should be assigned a family before inserting data into that particular column.
- Original big table also maintains some time-stamps for the data that can facilitate multiple versions of same data. For our assumption,we are maintaining latest three time-stamps in our big table.

- Boxwood project has some components that overlaps with big table.
- Sybase IQ and C-Store have have similar characteristics as both of them have shared nothing architecture and big table can leverage disk read performance benefits as they both can store data using column labels rather than row labels on the disk.

7 Conclusions

These are some of the major takeaways from this phase of project :

- We were successful in traversing through various components of DBMS.
- Gained knowledge on various aspects of Heap files, Index files and their functioning on the basis of different indexing strategies.
- We were also able to gain in-depth knowledge about the insertion of multiple records in a database based on different strategies.
- Efficiently implemented different strategies for insertion of records in batch and a single map in order to reduce disk accesses.
- Hands on experience with different types of indexing strategies such as indexing based on a single key and indexing based on composite keys.
- Successful insertion and deletion of maps in a heap file using a slot structure of HF-Page.
- Successfully optimized query retrieval with the help of different indexing strategies in order to reduce the number of disk I/O.
- Successfully implemented row join on multiple big tables using sort merge strategy
- Hands on experience with sorting techniques while implementing row sort in a big table.
- In the end,we were finally able to transform a relational database to a key-value pair providing flexibility to the end-user.

8 Appendix

- **Bhanu Preeti Anand :** Analysed and Implemented Row Sort, Map Insert.Helped in getCounts. Worked on Integration and issues related to functionality . Results, Interface Specification in Report. Testing various corner cases in Map Insert, Row Sort, Row Join, Get Counts.

- **Lakshmi Sindhuja Karuparti :** Building the structure of the database and implementation of Batch Insert. Brainstorm ideas on implementation of map insert. Testing of Batch Insert, Map Insert, Row Sort and Row Join. Contributed to report w.r.t Architecture and Batch Insert under implementation and interface specification.
- **Shantanu Aggarwal :** Implementation of row join with the help of Sort-merge functionality of Minibase, Testing of query retrieval using different index strategies, Information about row join in the final report
- **Sai Sruthi Mallineni :** Implemented Row Sort and Get Counts. Efficient implementation of Map Insert and Integration. Worked on issues with functionality and integration. Results analysed and put on Report. Testing related to multiple functionalities.
- **Sai Uttej Thunuguntla:** Implemented Map Insert, Query (Efficiency compared to file scan and index scan), Integration and issues in functionality. Implementation and interface specification related to Map Insert, Query in Report. Testing the final integrated code and solved issues in batch insert and row sort.
- **Srija :**Analysis and Implementation of Map Insert Logic, Analysis of Batch insert. Testing of batch insert,map insert,row sort and row join and working on report..

References

1. https://en.wikipedia.org/wiki/Bigtable
2. https://www.cs.rochester.edu/courses/261/spring2017/termpaper/16/paper.pdf
3. https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
4. https://ieeexplore.ieee.org/document/8892079
5. https://www.researchgate.net/publication/334122892_An_Efficient_External_Sort_Algorithm_Dufren
6. https://web.eecs.umich.edu/ manosk/assets/papers/bigtable.pdf
7. H. Chou and D. DeWitt, "An Evaluation ofBuffer Management Strategies for Relational Database Systems",Proc. of 11th VLDB Conf., August 1985
8. P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin,R. A. Lorie ,and T. G. Price. Access path selection in a relationaldatabase management system, 1979 ACM SIGMOD
9. Xuan Yang, Jianxiao Liu:Using Join Operation in Relational Database to Composite Web Services.
10. https://research.cs.wisc.edu/coral/minibase/minibase.html