Database Management System Implementation Phase 2

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

March 23, 2020

Abstract

The scope of the second phase of the project is to implement a Big Table like DBMS. This focuses on implementing various functionalities of a database management system for Big Table. Minibase (a database management system that is intended for educational use that was developed by Raghu Ramakrishnan) is considered as a base for this phase and changes relevant to Big Table are made on the top of it.The various functionalities that we implement would be storage mechanisms (heap files, secondary indexes based on B+ Trees), external sorting and a disk space management system. The objective of this phase is to provide the functionality of storing the data in the form of a key-value pair termed as Map in the Big Table which replaces tuple in a relational database.
Keywords: *Maps, B+ Tree, Buffer Manager, Heap Files, Sort, Conditional Expressions, Big Table, Stream*

1 Introduction

Minibase Distribution has a parser, optimizer, buffer pool manager, storage mechanisms (heap files, B+-trees as secondary indexes), and a disk space management system. Besides the DBMS, Minibase contains a graphical user interface and other graphical tools for studying the internal working of a DBMS.

With the rapid growth of third generation data models, there arouse a need for a data model that can accommodate flexible schema and organized distribution of data in columns in order to improve the performance. In addition to this, we require a data model that can deal with massive amounts of data(Petabytes) and can be much more scalable as compared to a relational model .

Big table claims to handle each and every given functionality.A record in a Big table is represented with the help of a map where key can be (row_name,col_name,time_stamp). Ordering of the data in a column can also be maintained with the help of a timestamp and simultaneously can boost up retrieval and insertion of records in the table. Big table works as a single-table database which is an ideal data model to implement google's search system as compared to multiple tables in a relational database.

1.1 Terminology

- Map : A key-value pair that is stored within a database.
- B+ trees : A tree based indexing structure basically used to range queries.Every internal node of a tree can have multiple records in the form of a key and a pointer.
- BigTable :This is a sparse, distributed, persistent multi- dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.
- Stream: This initializes maps in a specific order.
- Buffer Manager:This is responsible for bringing pages from physical disk to main memory as needed.
- Heap File:A heap file consists of set of unsorted records.
- Disk Manager :This performs the allocation and deallocation of pages in different files and also reads and writes pages from and to disk.
- External Sorting :External sorting is a technique in which the data is stored on the secondary memory, in which part by part data is loaded into the main memory and then sorting can be done over there. Then this sorted data will be stored in the intermediate files. Finally, these files will be merged to get a sorted data.
- Unclustered Index :The index and data are stored in different location in this index structure.

1.2 Goal Description

The goal of this phase is to implement the building blocks that will eventually use minibase operators to implement bigtable.

- The building block of the big table is a map which is achieved in a similar fashion as a tuple. The tuple consists of one integer,one string,one float and one char field whereas the map will consist of three strings (Row label, Column Label, Value) and one integer field(timestamp). Map is a fixed-length record.
- In the directory pages, data is stored in the form of a tuple in the minibase. Since tuple does not accommodate every attribute in Big Table, a new class Info is created to represent a directory page that stores the information of data pages.
- Each BigDB corresponds to a database which in turn contains a Big table with an indexing strategy. Each Big Table contains a heap file and index files built on the top of a heap file. Maps are inserted into the heap file via batch insert.

- In order to achieve the functionality of single and composite indexing as an advantage over a single field in the minibase, 5 big tables with respective indexing strategies are created.
- The advantage of bigtable over minibase lies in the sorting based on the priority of the fields rather than sorting on a single field.
- Data from the bigtable is retrieved using a query that lists the specific filters on a row, column and a value. The filters can either be null, single value or a range of values.
- Number of disk accesses for reads and writes are tracked for every operation, be it insertion of records or their retrieval.

1.3 Assumptions

- We assume that row label ,column label and value are String attributes while timestamp is always an integer
- We deal with fixed size records, with each string being 22 Bytes (20+2(delimiter)).
- Size of Map is fixed ie. 82 Bytes which also included the header field that contains metadata about the map
- Batch insert is achieved in a bottom-up manner ie. records are first inserted in the heap files followed by insertion of corresponding indexes in BTree files.
- We assume one to one matching in the database with one bigdb mapped to a single bigtable and one bigtable containing a heap file where the maps are stored.
- We are dealing with unclustured b+tree indexes only.
- We proceed with an assumption of timestamps being inserted in an ascending order.
- The time mentioned in all the tables under Results is in seconds.

2 Architecture

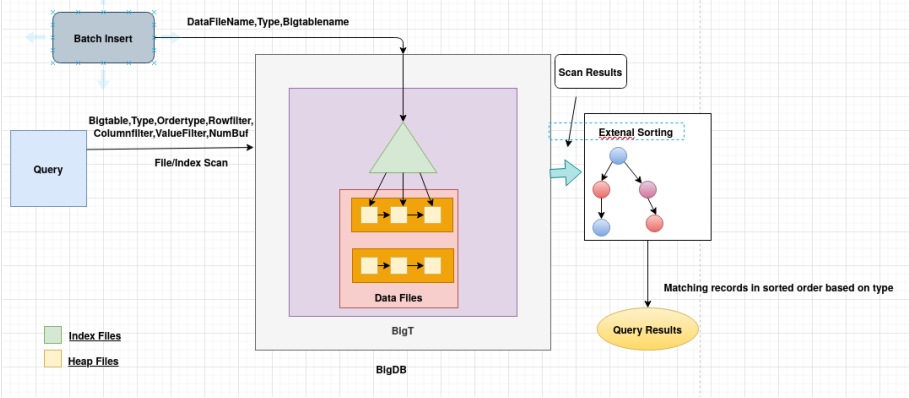

Figure 1: Architecture

- **HFPage** :It is a data organization where maps are stored. It creates a heap file page by extending the 'Page' object. Each HFPage of a heapfile is uniquely identified by its PageId and it also has pointers to the previous page and the next page. First 20 bytes of HFPage has metadata information about the page which consists of slot count, used Pointer, Free Space and Type. Data of the maps are organized in the fixed slots and the slot information is stored in the top of the page in the slot directory. Each slot stores the length of the map and the map offset which can uniquely identify a map. Maps are inserted from the back of the HFPage and slots are created from the top. When a map gets deleted the corresponding slot number stores invalid length and offset information so that new insertions in future can use these slot and data area in the page is always compacted. This is a simple data organization which is inherited by other components like BTSortedPage and BTreeHeaderPage.

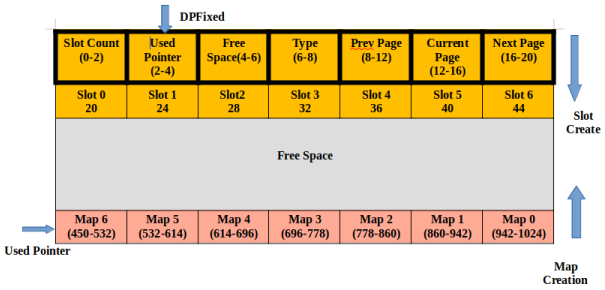

Figure 2: HF Page

- **Map** :Maps are the building blocks of a Big table. Data is stored in the form of a key-value pair where the key is a combination of row label, column label and timestamp whereas value is the actual data of a record.

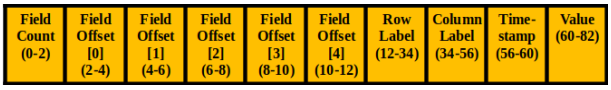

Figure 3: Map

- **MID** :Uniquely Identifies a map within a heap file by storing page id and slot number of a map in the form of (PageId,SlotNo).
- **Heap Files** :Heap files are a combination of directory pages and data pages(HFPages).

  DPInfo :

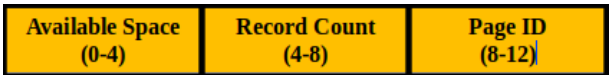

Figure 4: DPInfo

Each DataPageInfo object is an entry in the directory pages and can uniquely point to a HFPage in the heap file. It also consists of metadata information about available

space and record count in that HFPage.

Every heap file keeps track of directory pages and data pages allocated to it and performs inserts, deletes and the updates of the records efficiently.

- **BigT** :Big Table is a single table database which stores records in the form of maps with a key and value. Each BigT consists of a heap file and the index files that are created based on the input type specified. Whenever a map has to be inserted in a bigt, it is inserted in the heap file present in the bigt and an index key is inserted into the index files as well. It provides interfaces for getting the total number of maps, unique row labels and column labels present in it. In a given big table at any point of time we maintain only three maps with the same row and column labels but different timestamps.

- **BigDB** :It is a single database with a one to one mapping with bigT.

- **Batch Insert** :This functionality creates the bigt with the bigtable name from the datafilename. In case, the bigt already exists maps are inserted into the existing big table. numbuf is the number of buffers allocated during the batch insert. Type handles different kinds of indexing strategies.

- **Query Retrieval** :This functionality access the bigdb and retrieves the maps that matches the criteria specified by the various filters. ordertype indicates the order in which the results are displayed.

- **Sort** :This aims at sorting the records retrieved from the scan. Here, external sorting is used that sorts the massive amounts of data records. This kind of sorting is preferable in case of data that cannot fit in the main memory.

3 Implementation

3.1 Maps

A map class is implemented that is similar to a tuple in Minibase. A map is a byte array of size 82 where initial 12 bytes have been reserved for metadata that stores field count and an array of field offset, while the next 70 bytes depict the data in the map.

- **Attributes** :

  private byte[] data; (Data of the map stored in the form of bytes)

  private int map_offset; (start position of this map in data[])

  private int map_length = 82;

  12 [fldCnt (2 bytes), fldOffset[4] (10 bytes)) + 22(row label length) + 22(column label length) + 4(timestamp length) + 22(value length)

  private short fldCnt = 4;

(Since the map is fixed and only has 4 fileds rowLabel, columnLabel, Time Stamp and Value)

private short[] fldOffset;

(stores the offsets(starting address) of the map fields)

- **Methods**:

  public void setHdr():

  Initializes the meta data required for each map object. Copies the fieldCount into first 2 bytes of data[] array and initializes the fieldOffset[] array with appropriate values based on lengths of the field and copies this offset information into next 10 bytes of the data[] array.

  public Map setRowLabel(String val) :

  Sets the rowLabel of the map to a given value by converting it to bytes and allocating 12 to 34 bytes of the data[] array.

  public Map setColumnLabel(String val) :

  Sets the columnLabel of the map to a given value by converting it to bytes and allocating 34 to 56 bytes of the data[] array.

  public Map setTimeStamp(int val) :

  Sets the timestamp of the map to a given value by converting it to bytes and allocating 56 to 60 bytes of the data[] array.

  public Map setValue(String val):

  Sets the value of the map to a given value by converting it to bytes and allocating 12 to 34 bytes of the data[] array.

  public String getRowLabel():

  Returns the rowLabel of the map by converting the byte sequence stored from 12 to 34 in the data array to a String value.

  public String getColumnLabel():

  Returns the Column Label of the map by converting the byte sequence stored from 34 to 56 in the data array to a String value.

  public int getTimeStamp():

  Returns the TimeStamp of the map by converting the byte sequence stored from 56 to 60 in the data array to an Integer Value.

  public String getValue():

  Returns the Value of the map by converting the byte sequence stored from 60 to 82 in the data array to a String value.

  public byte[] getMapByteArray():

Copies the data[] array into a temporary byte[] array and returns it.

public void print():

Converts the byte sequence in data[] array and prints the map fields in the following format

(rowLabel,columnLabel,timeStamp)-¿ value

public int getLength() / public short size():

Returns the length of the map which is 82.

public void mapInit(byte[] amap, int offset, int length)

Initializes the map with given byte array from the given offset and also initializes length of the map.

public void mapSet(byte[] record, int offset, int length)

Sets a given map with the record bytes using the array copy method from the given offset and assigns the length to the map length.

public void mapCopy(Map fromMap)

Copies the fromMap to the current map using array copy method by converting map to bytes.

3.2 BigT

- **Methods**

  public BigT(String name, int type)

  Creates a Heap file with given name and based on the type no index, 1 index or two index files are created.

  void deleteBigT()

  Deletes the heap file and all the relevant index files based on type.

  int getMapCnt()

  Returns the the number of records stored in the heap file by making a call to getRecCnt() of heap class which internally traverses through all the directory page entries and sums up the records present in each HFPage.

  int getRowCnt()

  Return the number of unique row labels in the given bigtable. After the records are inserted in bigt, a file scan object is instantiated on the given bigt file with conditional expressions set to null so that all maps in the file are retrieved. This file scan object is then passed as input to Sort() class with orderType as 1, so all the records are sorted based on rowLabel first and then on columnLabel and value. Then the unique row values retrieved by sort iterator are counted and returned as result.

  int getColumnCnt()

Return the number of unique column labels in the given bigtable. After the records are inserted in bigt, a file scan object is instantiated on the given bigt file with conditional expressions set to null so that all maps in the file are retrieved. This file scan object is then passed as input to Sort() class with orderType as 2, so all the records are sorted based on columnLabel first and then on columnLabel and value. Then the unique row values retrieved by sort iterator are counted and returned as result.

public Map constructMap(byte[] mapPtr)

Constructs a map with given byte array as input and initializes the header data of the map using map.setHdr() function and returns it.

public MID insertMap(byte[] mapPtr)

Inserts a map into the heap file instantiated in the constructor. The heap file inserts are happened by calling 'insertRecord(mapPtr)' function which traverses the directory pages of the heap file to find a HFPage with free space greater than the size of the map and inserts into an empty slot in that HFPage or creates a new slot. The MID of the inserted map is returned.

public void insertIndex()

After all the records are inserted into the heap and duplicates based on versioning are removed, a scan on the heap file is initialized and an index key is formed from each map retrieved from scan based on the index type and these index keys are inserted into appropriate index files based on the type of BigT.

public Stream openStream(int orderType, String rowFilter, String colFilter, String valFilter, int numbuf)

Initialize a stream of maps where row label matching rowFilter, column label matching columnFilter, and value label matching valueFilter.

3.3 BigDB

This class creates and maintains a BigTable of maps and btree based index files. This class enables user to create, open, close, and destroy a database.

- **Functions**

  public BigDB(int type)

  It enables you to specify an indexing scheme which applies to all BigT data files in the database and is initialised to the type attribute.

  Since BigDB has one to one mapping with the Big table, upon creation of the database, a big table is created. In big table , the maps are organised in the form of heap files, with the help of insertMap(). Later the indexes are constructed over the heap files using insertIndex(). Upon closure of the program, all the database, heap files and index trees are destroyed using the destroyDB().

3.4 Versioning

After the batch insert operation is performed on a given big table, there can only be at most three maps with the same rowLabel and columnLabel. This is implemented in removeDuplicates() method of BigT class by creating a temporary utility index which stores rowLabel+columnLabel of a given map as index Key. Once insertions are completed into the given bigt, all the corresponding indexKeys are inserted into this temporary utility index file. After populating this temporary index file, indexOnly query is performed and all the index Keys are retrieved by calling iscan.get_nextMidPair(), when the count of a unique index key which is a combination of rowLabel+columnLabel exceeds 3, the MID with least time stamp is removed from the heap file. Once all the duplicates are removed this temporary index file is deleted by calling destroyFile() function of BtreeFile class.

3.5 Indexing Schemes

A BigDB is instantiated based on the type of index and five types of indexing schemes are supported.

- **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) and one btree to index timestamps
- **Type 5** : One btree to index row label and value (combined key) and one btree to index timestamps

When a batch insert is performed all the maps are inserted into bigt, and duplicates based on versioning are removed and then the index trees are built on the updated BigT file by scanning the maps present in it. Based on BigT type, a Btree File is instantiated with name, key type and length of the key.

- **Type 1** : No index File is created.
- **Type 2** : A Btree Index File with file name bigtFilename+"Index0" is created. Key type is given as String and keyLength is given as 22 (default Map Row Label length). Since indexing is based on row labels, index key is constructed by calling the getRowLabel() method of the map class and this key is inserted by calling insert() function of BTreeFile class.
- **Type 3** :A Btree Index File with file name bigtFilename+"Index0" is created. Key type is given as String and keyLength is given as 22 (default Map Column Label length). Since indexing is based on column labels, index key is constructed by calling the getColumnLabel() method of the Map class and this key is inserted by calling insert() function of BTreeFile class.

- **Type 4** : In this BigT type, two index files are created with names bigtFilename+"Index0" and bigtFilename+"Index1"

  The first index file has a combined key based on column and row label so the index key is created by concatenating column and row labels of a map. Key Type is given as String and the length of the key is given 44. After consstructing the index key by calling map.getRowLabel()+map. getColumnLabe() methods of map the concatenated key is inserted using insert() function of BTreeFile class.

  The second Index File is created based on timestamp so the KeyType is given as Integer and key Length is given as 4. Index Key is obtained by calling getTimeStamp() method of Map class and inserted into the index file by calling insert() function of BTreeFile class.

- **Type 5** : This is similar to Type 4 Indexing as mentioned above but the first Index file key is constructed by concatenating rowLabel and value fields of a map.

A Btree Index File with file name bigtFilename+"Index0" is created. Key type is given as String and keyLength is given as 22 (default Map Column Label length). Since indexing is based on column labels, index key is constructed by calling the getColumnLabel() method of the Map class and this key is inserted by calling insert() function of BTreeFile class.

In type 2 and type 3, the keys are row and column label respectively. But type 4 and 5 have combined keys. To obtain a combined key, the row and column labels are concatenated in type 4. While in type 5, the combined key is obtained with concatenation of row label and value. Type 4 and type 5 comprise two index trees. The second index tree is on time stamp in both the types.

After the creation of Btree Files, the key are obtained by scanning and inserted into the Btrees.

3.6 Info Class

A data record is stored in the form of a map which has 4 fields that are of fixed length. A directory entry of a DataPage Info class stores 3 fields available space, record count and page id of a data page which cannot be stored in the form of maps.

Therefore a new class, Info is created to define the structure of an entry in directory page which can be used in DataPage Info class, with the following specifications:

- Methods

  Info() : In this constructor, we initialize info object with default values if no arguments are passed else with the values of the arguments. We can also create an info object from another info object when that object is passed as an argument.

  byte[] getInfoByteArray(): it returns the byte array that contains the information of info(excluding meta data) by extracting from data array.

int getIntFld(int fldNo) : given the field number, it gives the actual value of that field.

Info setIntFld(int fldNo, int val): sets the given value into the given field number of the data array.

void setHdr(short numFlds, AttrType types[]): This sets(allocates sizes) the header information for an info object. This function sets the following:

1.field Count

2. field offset array. It has 4 values which are offsets of availspace, recct and pageId from DataPageInfo object, the last value being the end of the info object.

3. actual info object array.

void print(): This gets the actual values from data array by converting the bytes in the data array to integers and finally prints the info data.

3.7 Streams

This class initializes a stream of maps based on the filters provided by the user. This method either uses file scan or an index scan depending upon which seems to be advantageous and quick. It also retrieves the maps according to the user specification. It is similar to a scan class in a minibase but with additional kinds of access to bigtable. This in turn uses get_next() and closeStream() to achieve this functionality.

- **Methods**

  Stream (BigT bigT, int orderType, String rowFilter, String colFilter, String valFilter, int numbuf):

  In the constructor of Stream class, bigt is initialized followed by the scan and sort based on DB type. Based on the type of scan, the conditional expressions are constructed from filter () or indexFilter () function. Scan is initialized using IndexScan class that takes in 2 different kinds of conditional expressions based upon which indexing and file scan is performed. The results of scan are taken by the Sort that returns the maps in the sorted order as per the required type.

  IndexScan Vs Filescan based on type of database and indexed files.

  As type 1 takes in no indexes, a file scan is created irrespective of the row, column and value filter.

  Type 2, being a row indexed database, Index scan is found to be efficient if row filter has a single value or a range of values. So, an index scan is created in this case, else a file scan is instantiated.

  Similarly, for DB type 3 with column based indexed file, the corresponding scan is instantiated that helps in easy and quick retrieval.

  In case of type 4, indexes are on composite key (row and column label) and time stamp, but index on timestamp seems not to be useful in case of retrieval. Index scan

is used in case both row filter and column filter are not * as it retrieves results quickly. This scan seems to be of no use in case of one filter being null as it returns every map leading to no advantage of an index file.

Type5, similar to type 4 a file scan or an index scan is performed according to the users requirement.

Map getNext (MID mid)

It internally calls the sort.get_next(), which sorts the retrieved maps from the scan in ascending order.

void closestream ()

The iterator is closed and all the temporary files and the buffer pages used for sorting are deallocated using sort.close() function internally.

Other Important Functions

Below are the functions modified according to the functionality for the Stream class:

A function compareMapToMapForSorting is created in the maputils that serves as a comparator for user defined object Map. This function has been modified according to the order type in the query. This handles the composite key indexed files by concatenating both the required fields and comparing towards the other map.

The constructor in the index scan is overloaded with parameters indSelects and evalSelects so as to handle the conditional expressions for index scan and filtering respectively. This does not use indexonly queries so as to retrieve the entire map.

3.8 MapUtils

This class has functions related to map comparisons and equality of maps.

- **Functions**

  CompareMapWithMap(AttrType fldType, Map m1, Map m2, int map_fld_no)

  This function is used for comparing two maps. This is used to compare the map based on a specific field. This is modified for indexing purpose by concatenating fields in case of a composite key.

  CompareMapWithMapForSorting(AttrType fldType, Map m1, Map m2, int map_fld_no)

  This function is used for comparing two maps specifically used for sorting. This is used to compare the map based on a specific field. This is modified for sorting purpose by comparing the fields according to the priority.

  equal(Map m1,Map m2

  This function Compares two Map in all fields.

  setValue(Map m1,Map m2, int fldNo, AttrType fldType

  This function set up a map in specified field from a map.

3.9 BigDB

This class creates and maintains a BigTable of maps and btree based index files. This class enables user to create, open, close, and destroy a database.

- **Functions**

  public BigDB(int type)

  It enables you to specify an indexing scheme which applies to all BigT data files in the database.

3.10 Pcounter

The Pcounter class keeps the track of page reads and writes during query and batch insert with the help of static variables rcounter and wcounter. They are static as they should be common to all reads and writes.

- **Attributes**

  public static int rcounter =0

  It is static variable which maintains a track of number of page writes.

  public static int wcounter = 0

  It is a static variable which maintains a track of number of page reads.

  readIncrement and writeIncrement are called in read_page() and write_page() respectively present in BigDB class. The read_page and write_page are called when a page is read or written to disk.

3.11 Batch Insert

Query Format

batchinsert DATAFILENAME TYPE BIGTABLENAME

- **Insertion into bigT**

  The given data file is read line by line by using BufferedReader class and map is constructed from each line by splitting it based on "," and populating all the fields based on the ordering of the values in the given test file. The byte array of this map is obtained by calling getMapByteArray() of the map class. This byte array is passed to insert() function of bigdb class, where the map is inserted into the corresponding BigT class. This process is repeated for all the lines in the given data file and the corresponding maps are inserted into the Big Table.

- **Removing Duplicates from bigT**

  As mentioned in section 3.3, once the insertions are performed a temporary index file is created and all the duplicates are removed from the big table by performing an index scan and in the end this temporary index file is destroyed.

- **Populating Index Files**

  When all the duplicates are removed in step 2, scan is performed on updated heap file and all the appropriate index files are populated as mentioned above.

- **Displaying the page counts and time taken**

  Once the batch insert operation is over time taken for all the above steps is displayed. Number of disk pages that were read and written are also printed by calling rcounter and wcounter attributes of Pcounter class.

3.12 Query Retrieval

Query Format

query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF

In bigtable,the maps are retrieved based on different order types. Based on the users requirement, we use appropriate index scan or a file scan to retrieve the results. Stream class in bigt package is used to retrieve the maps in the specified order. It is similar to a scan class in a minibase but provides different kinds of access to bigtable. Here, we use get_next() and closestream() functions to achieve this functionality. The row, column and value filter can be a single value, range or '*'. Based on a combination of DB type and order type index and file scans are created respectively.

- **Stream (BigT bigT, int orderType, String rowFilter, String colFilter, String valFilter, int numbuf)** In the stream constructor, bigt is initialized followed by the scan and sort based on DB type. Based on the type of scan, the conditional expressions are constructed from filter () or indexFilter () function. In DB type 1, there are no indexes, so a file scan is created irrespective of the row, column and value filter. Type 2, being a row indexed database, is used to retrieve maps quickly if the row filter has a single value or range of values. So, an index scan is created in this case, else a file scan is instantiated. Similarly, for DB type 3, a column index is created and the corresponding scan is instantiated that helps in easy and quick retrieval. In DB type 4, indexes are created on composite key (row and column label) and time stamp, but index on timestamp seems not to be useful in case of retrieval. Since, a composite index is used, if both the filter values aren't '*', the index scan is used for faster retrieval. This scan seems to be of no use in case of one filter being null as it returns every map leading to no advantage of an index file. Type5, similar to type 4 a file scan or an index scan is performed according to the users requirement.

- **Map getNext (MID mid)** In getNext, the maps are returned in a specified order. It internally calls the sort.get_next(), which sorts the retrieved maps from the scan in ascending order.

- **void closestream ()** The iterator is closed and all the pages used for sorting are deallocated using sort.close() function internally.

4 System requirements and Execution Steps

- 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, query, getRowcount or getColumnCount 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. datafilename (path of the file) 3. type (type of db and indexing schemes) 4. databasename (name of the database) 5. numbuf (number of buffers)

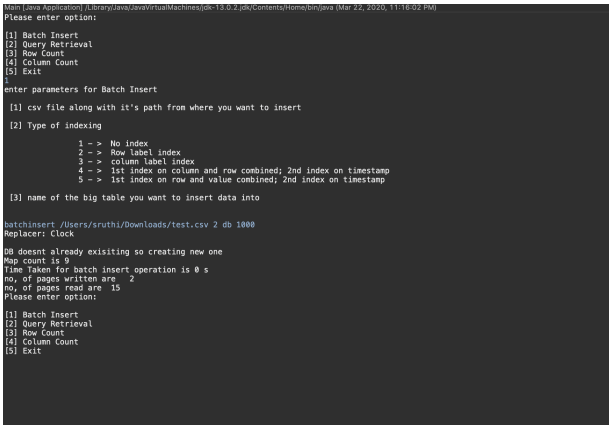

Figure 5: Batch Insert

- On choosing query, enter the query operation with the following parameters. 1. query (type of operation) 2. databasename (name of the database) 3. type (type of db and indexing schemes) 4. ordertype (sort order of the results) 5. rowfilter (filter on the row either *(no filter), range of values inserted in square braces, individual value) 6. colFilter (filter on the column) 7. valueFilter (filter on the value) 8. numbuf (number of buffers)

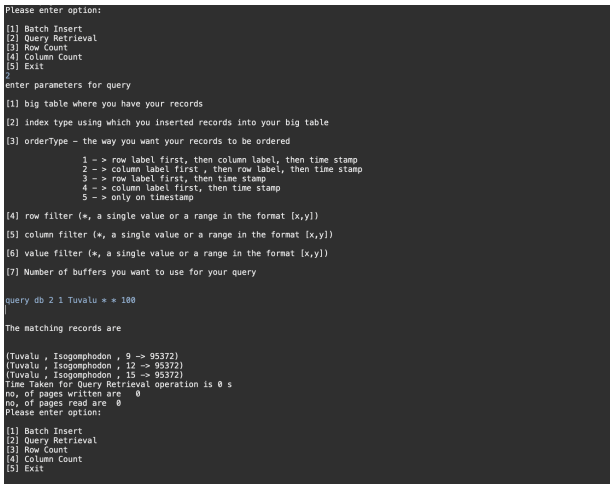

Figure 6: Query

- Insert number of buffers for getting row count and column count

5 Results

5.1 Batch Insert

Type	time taken	read	write	buffers	Number of Maps
1	20	220250	7539	1000	20000
1	18	667077	11799	500	20000
1	19	847694	15758	300	20000
1	19	919329	18930	200	20000
2	21	218369	7966	1000	20000
2	20	668247	12684	500	20000
2	19	848252	17000	300	20000
2	19	921615	21710	200	20000
4	22	218694	8690	1000	20000
4	21	672856	16572	500	20000
4	21	856205	24602	300	20000
4	20	931436	31043	200	20000

Figure 7: Batch Insert

A batch insert operation is performed on a data file with 200000 records and the above table depicts the number of reads, writes and time taken as the number of buffers changes for the databases with no, single and two indexes. If the DB type is 1, then batch insertion for 1000 buffer pages takes 20 seconds , the number of pages read and written are 220250 and 7539. As the number of buffers are reduced to 500 and 300, the reads and write increase.

For DB type 2, an index tree is constructed over the row label, the batch insertion happens in 21 seconds for 1000 buffer pages, the number of pages read and written from the disk are 218369 and 7966 respectively. Similarly it is observed that the number of reads and writes increase as the buffer pages decrease. While it can be observed that the number of writes for DB type 2 is more than the DB with no indexes for 1000 buffer pages as an construction of an index tree leads to more reads and writes.

In Db type 4, the batch insertion is completed in 22 seconds for 1000 buffer pages. While the number of reads and writes are 218694 and 8690 respectively. Similar to DB type 1 and

2 , the reads and writes increase drastically as the number of buffers decreases. It can be seen that the number of writes for DB type 4 is more as another index tree based on time stamp is constructed.

It can be concluded that the number of buffers is inversely proportional to the number of reads and writes.

5.2 Multiple Batch Insert

Type	time taken	read	write	buffers	Number of Maps
1	18	220265	7539	1000	20000
1	82	8573031	34883	1000	20000
1	25	2926008	26597	1000	10000
2	20	218369	7966	1000	20000
2	92	8655845	36300	1000	20000
2	31	3052205	28389	1000	10000
4	21	218694	8690	1000	20000
4	83	8805876	40734	1000	20000
4	30	3104022	33244	1000	10000
3	18	218369	7947	1000	20000
3	91	8650107	36290	1000	20000
3	29	3056407	28289	1000	10000
5	19	218476	8418	1000	20000
5	76	8757624	39588	1000	20000
5	25	3131370	32414	1000	10000

Figure 8: Multiple Batch Insert

The above table shows the time taken, number of reads and writes during a multiple batch insertion for various kinds of DB when the number of buffer pages are 1000. In DB type 1, no index tree, the time taken , reads and writes for the first 20,000 maps are 18 seconds, 2,20,265 and 7539 respectively. As other 20,000 maps are inserted the time taken along with reads and writes increase drastically as the number of duplicates to be removed are high due to the versioning property, to have three maps at maximum. The duplicates are removed after the entire construction of the heap file and a temporary Btree is constructed to maintain versioning. While the time taken for insertion , reads and writes for the last 10,000 maps are less. It is due to less number of duplicates and maps. Similar trends can be observed in other DB types. It can be concluded that the read, writes and time taken for insertion depends on the duplicate maps and the amount of data for the proposed design.

5.3 Query Performance

Type	Ordertype	Row filter	col filter	Val filter	Num buf	time	read	write
1	1	*	*	*	300	2	6238	2981
1	1	Tuvalu	*	*	300	0	3505	6
1	1	*	Swan	*	300	0	3502	3
1	1	*	*	[20,5000]	300	0	3576	65
1	1	Tuvalu	Swan	15198	300	0	3601	4
2	1	*	*	*	300	2	6240	3151
2	1	Tuvalu	*	*	300	3	610746	22
2	1	Tuvalu,Zambia	*	*	300	32	6783815	453
2	1	Tuvalu	Swan	*	300	3	610320	14
2	1	Tuvalu	Swan	15,198	300	3	610212	14
3	2	*	*	*	300	3	6240	3150
3	2	*	Swan	*	300	3	576959	21
3	2	*	Raven,Swan	*	300	33	7442452	508
3	2	Tuvalu	Swan	*	300	2	576942	13
3	2	Tuvalu	Swan	74935	300	2	576583	13
4	1	*	*	*	300	2	6239	3525
4	1	Tuvalu	*	*	300	0	3523	10
4	1	*	Swan	*	300	0	3511	3
4	1	Tuvalu	Swan	*	300	0	6330	3
4	1	Tuvalu,Zambia	Rave,Swan	*	300	0	3518	3
4	1	Tuvalu	Swan	15198	300	0	6330	3
5	1	*	*	*	300	2	6243	3505
5	1	Tuvalu	*	*	300	0	3522	11
5	1	Tuvalu,Zambia	*	*	300	0	3789	262
5	1	Tuvalu	Swan	*	300	0	3504	5
5	1	Tuvalu	Swan	15198	300	0	2280	3

Figure 9: Query Retrieval

The above table displays time taken, number of reads, writes for various DB's with different order types and filters for 300 buffer pages. According to the design, if the DB type has an index and if the filter is on the index column then an index scan is instantiated else a file scan is created.

For Type 1 DB, since a file scan is performed, the time taken to retrieve maps based on filters are in the range of 0-2 seconds, a file scan is performed on it .

For Type 2 DB, an index tree is constructed on a row label, so the row filter with single value happens in 3 seconds while range search takes 32 seconds and number of reads is 6783815, which is 10 times more than other filters. Range search in an unclustered index is expensive as the scan cost is BD(R + 0.15) and every element in the range must perform a full scan. This drastically increases the time taken, read and write counts.

In Type 3 DB, an index tree is constructed on column label, the retrieval time if the column filter is single value is 3 seconds while the range scan takes 33 seconds and number of reads is 74,42,452. A similar observation to range search in type 2 DB is noted here.

In Type 4 DB, a index tree on combined key (row and column label) is formed, the retrieval time of maps for various filters is really quick as an index only query can be performed on the tree.

Similar to Type 4 DB, in Type 5 DB, an index tree on combined key (row label and value) is formed, the map retrieval for various filters is again really fast due to index only queries.

It can be concluded that an index scan can be really helpful with queries that have filters on keys otherwise a file scan can be used.

6 Discussion and Conclusions

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

- We were successful in traversing through various components of DBMS.
- Understood how a tuple in a Relational DBMS can be mapped to a map in key-value store.
- 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.
- 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 of maps in a heap file using a slot structure of HFPage.
- Successfully optimized query retrieval with the help of different indexing strategies in order to reduce the number of disk I/O.

References

- http://dbmsfortech.blogspot.com/2016/05/buffer-management.html
- https://www.includehelp.com/algorithms/external-merge-sorting.aspx
- https://research.cs.wisc.edu/coral/mini_doc/minibase.html
- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber:Bigtable: A Distributed Storage System for Structured Data

7 Appendix

1. Shantanu Aggarwal - Working of Heap files, Maps, Initialization of BigT for 5 types of indexing, Impelementation of BigDB, Integration and testing of batch insert and query retrieval, Heap files in architecture section of report,Modification of BTIndex pages as per Bigtable architecture
2. Sai Uttej - Analysed Heaps, modified HFPage. Implemented BigT, Versioning for Insert Maps, Modified Index Scan iterators.Implemented Batch Insert logic. Resolved issues in Streams. Modified IndexTest and SortTest to test the functionalities in BigTable architecture. Changed PredEval class for index and sort iterators. Complete Integration, Testing and report
3. Bhanu Preeti Anand - Implemented HeapFile, Implemented Streams. Implemented maputils, query logic and main class. Resolved issues in Index Scan,Complete Integration, Testing. Worked on implementation, results in report.

4. Sai Sruthi Mallineni- Implemented HeapFile, Analysed sort and implemented as per the requirement. Implemented maputils, query logic and main class. Resolved issues in bigdb, Complete Integration, Testing. Worked on report.

5. Lakshmi Sindhuja Karupati - Analysed Map class, implemented Info class and DataPageInfo class. Modified IndexScan and Index Test. Implemented clustering schemes for 5 orderTypes. Worked on Integration and testing of all functionalities as well of report writing.

6. Srija Saha- Analysed Map Structure,Implemented heap file directory page info,Modified HFTest class to test the heap file implementation,Modified indexScan and IndexTest,Implemented Sort Logic for different order types ,Worked on Integration and testing of batch insert and query retrieval and report writing.