CSE 510: Database Management System Implementation

Phase 2: Analysis and Implementation of Big Table DBMS with the help of Minibase Modules.

GROUP 2:
Aashka Bhavesh Parikh
Maneesha Guntur
Himali Hemant Gajare
Rigved Alankar
Kshitij Dhyani

ABSTRACT
Minibase was originally implemented as an RDBMS. The objective of the phase 2 report is to describe the steps involved in using Minibase modules to implement a Big table DBMS. To achieve this, we modified the tuple-based implementations in Minibase into map-based implementations. These changes were made in the Minibase modules and three main classes - BigT, Map, and Stream. Once a Bigtable-like DBMS was achieved, we defined five different indexes and analyzed read/write operations and the time taken to process different queries. This resulted in gaining a general understanding and analytical perspective towards indexing techniques and the Big table DBMS as a whole.

***Keywords*:** DBMS, RDBMS, Big Tables, Buffer, Read, Write, Minibase, Map, Tuple, Index, Query, Streams, Heap File, B Trees, B+ Trees, Batch, Queries, Key-Value, IndexFiles.

INDEX

INTRODUCTION

TERMINOLOGY

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

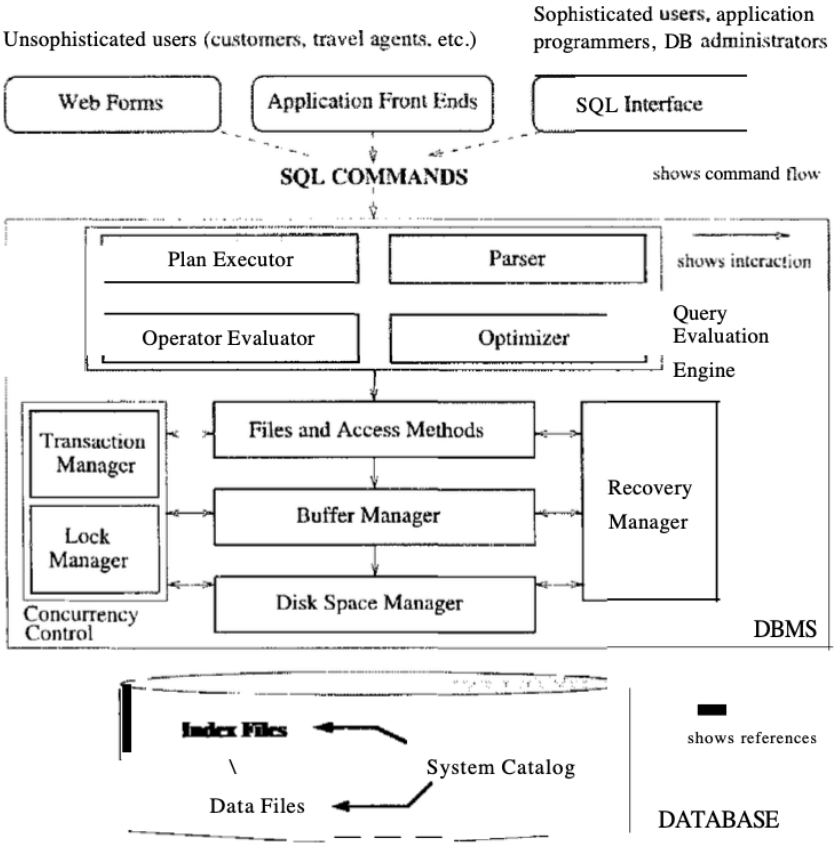


Figure 1. Architecture of DBMS

Maps:
Maps is a key-value pair type of datasets that enable associating data values to a set key. The set keys can be used for identifying the data values. With reference to big table

architecture within the Java Minibase, Maps have a combination of 3 keys that point to a single String value.

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

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

B+ Trees:
B+ Trees are also an advanced variation of self-balancing trees, where all data points are leaf nodes [2]. The B+ tree is a highly utilized index in modern database systems. Each node in the tree, represented as a disk page, typically has a size of 4096 bytes. The leaves of the tree hold the database records, while index pages located above the leaf level contain only index records. The search for a single database record involves accessing exactly one node at each level of the tree. Due to its organization of clustering data in leaf pages in the order of the indexing attributes, the B+ tree is optimal for key-range searches. Additionally, the B+ tree is dynamically organized, ensuring that the height of the tree remains balanced even after record insertions or deletions [3].

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

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

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

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

GOALS

Minibase originally implements various concepts of relational database management systems, the main aim of phase 2 of this project is to transition these implementations into that of a Bigtable-like DBMS. This is achieved through making modifications to various classes such as BigT, stream, map, heap package, disk manager, and iterator package etc. Additional objectives include developing a batch query that allows inserting data in a batch to the DB and also analyzing various queries on the basis of read-write operations performed when index type and buffer number are tuned to different values.

ASSUMPTIONS

1. The RowLabel, Column Label, and Value fields are categorized as 'String' (Java.lang.String).
2. The TimeStamp field is assumed to be of Integer type.
3. The default length of the RowLabel, Column Label, and Value fields is 20 bytes. If the length of any field exceeds it will throw an error.
4. If the size of the Map exceeds the page size(1024 Bytes) it will throw an error.
5. If the program is terminated and restarted, the previously inserted records must be re-inserted since there is no cache maintained for them.
6. It is assumed that The Range filters are closely packed in the query, without any whitespace between the range values.

SOLUTION

Task 1: BigT Class Implementation

BigT - bigt

The bigt class maintains all the heapfiles and indexfiles required to implement different indexing and clustering strategies.

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

1 - No indexing
2 - Index row labels
3 - Index column labels

4 - Index column label and row label (combined key)
5 - Index row label and value (combined key)

The Bigt class has two different constructors:

1. `bigt(String name, int type)`: Creates a new instance for the bigt class. It takes the name of the table as an input and the indexing type. It also creates the names of the indexfiles and the heapfiles. Later, it creates the instances of BTreeFile for all the indextypes.
2. `bigt(String name)`: This constructor used during the Query Operations. It is similar to the previous constructor however it does not create the indexfiles.

The following getter/setter methods are used to set or get different field values of the bigt.

1. `int getMapCnt()`: Return number of maps in the bigtable.
2. `int getRowCnt()`: Return number of distinct row labels in the bigtable.
3. `int getColumnCnt()`: Return number of distinct column labels in the bigtable.

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

1. `MID insertMap(byte[] mapPtr)`: Using the bigt object, the insertMap method is called which internally executes the insertMap methodology of the HeapFile class.
2. `void insertIndex(MID mid, Map map, int type)`: Inserts the map into the corresponding BTreefiles (indexfiles).
3. `void deleteBigt()`: Deletes the bigtable from the system.
4. `Stream openStream(int indextype, int orderType, String rowFilter, columnFilter, valueFilter)`: This function initiates a Scan object that will include the rowFilter, columnFilter, valueFilter if any and then iterate over the indexes and files to locate the records.

BigT - Map

The Map class is very similar to the Tuple class. In the original implementation of Java minibase, records were maintained inside a Tuple. Whereas in the bigtable implementation, each record is maintained like a Map. The size and structure of the map is fixed and can't be modified.

Every map has the following four fields and datatypes:

RowLabel: attrString
ColumnLabel: attrString
TimeStamp: attrInteger
Value: attrString

The keys of the map are the “RowLabel”, “ColumnLabel” and “TimeStamp”. The data stored in the “RowLabel”, “ColumnLabel”, and “Value” fields, all of which are of type String, may vary in size depending on the length of the data. In contrast, the "TimeStamp" field is a fixed size of 4 bytes, as it is an integer field. As the implementation of the Map is comparable to that of the Tuple, many of the functions are also quite similar.

The Map class has four different constructors:

1. `Map()`: Creates a map with blank data, default length and default offset.
2. `Map(Map fromMap)`: Creates a map from another map.
3. `Map(int size)`: Creates a blank map with length indicated in the parameter.
4. `Map(byte[] amap, int offset, int len)`: Creates a map with a bytearray (amap) mentioned in the parameter of size “len”.

The following getter/setter methods are used to set or get different field values of the Map.

1. `String getRowLabel()`: Returns the row label.
2. `String getColumnLabel()`: Returns the column label.
3. `int getTimeStamp()`: Returns the timestamp.
4. `String getValue()`: Returns the value.
5. `Map setRowLabel(String val)`: Sets the row label.
6. `Map setColumnLabel(String val)`: Sets the column label.
7. `Map setTimeStamp(int val)`: Sets the timestamp.
8. `Map setValue(String val)`: Sets the value.

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

1. `void setHdr(short numFlds, AttrType types[], short strSizes[])`: Takes the number of fields as an input of the function along with the attribute types and sizes and accordingly generates a header for the entire Map and maintains it in the fldOffset array which is referenced at multiple places within the Map. This header provides field differentiators within a byte array to separate different fields of the map. By default, the numFlds in a map is 4.
2. `byte[] getMapByteArray()`: Retrieves the byteArray of a map.

3. `void print()`: Prints the map to the console. This method is mainly accessed in the Stream class while querying the data.
4. `short size()`: Returns the size of the map in bytes. It uses the map_offset (starting position of map) and the fldOffset to extract the size.
5. `void mapInit(byte[] amap, int offset, int len)`: It is similar to the Map constructor but is only referenced when the constructor is not to be used.
6. `void mapCopy(Map fromMap)`: Extracted the byte array of the fromMap and copies it to the referencing map.
7. `void mapSet(byte[] frommap, int offset, int len)`: Set a map with the given byte array and offset.
8. `void mapSet(byte[] frommap, int offset)`: Set a map with the given byte array and offset.
9. `void setFldOffset(byte[] mapByteArray)`: Sets the field offset for the given byte array of a map by maintaining sizes of different fields.

BigT - Stream

This class is analogous to heap.Scan. The stream class allows the user to iterate over the maps present in the bigtable. The class employs the `getNext` method to iterate over the maps in the bigtable. A switch statement is used to determine how the filter condition is created based on the index type and row/column/value filters. It also has the function `closestream` function to close the stream. We used an object of type CondExpr to store the value of the final condition expression for filtering.

The Stream class has following functions:

1. `Stream(String bigtName, String indexFilename, int indexType, int orderType, String rowFilter, String columnFilter, String valueFilter, int numBuf)`: It accepts parameters namely the bigtable which specifies the bigtable whose elements we want to iterate over, index type which is an integer ranging from 1 to 5 specifying the indexing and clustering strategy used to store the elements, orderType and the row, column and value filters, the numbuf and the file to operate on.
2. `void closestream()`: Closes the stream object.
3. `Map getNext(MID mid)`: Retrieve the next map in the stream.
4. `CondExpr[] getKeyFilterForIndexType(int indexType, String rowFilter, String columnFilter, String valueFilter)`: Returns a filter condition expression based on the indexType which will be useful in fetching the keys while implementing a scan involving both the index files and the heap files.

5. `List<CondExpr> processFilter(String filter, int fldNum):` The filters are processed using this function wherein the operands and operators are fetched from the expressions which are then added to the a list of type CondExpr which is a linked list implementation of storing all the conditions connected by an OR operator.

Below listed are the operations that happen for each index type respectively.

- Type 1: We need to iterate through all the maps and we use the FileScanMap method which will scan the contents of the entire heap file.

For all other cases we use the MapScan Class which includes the filter condition expression as an input parameter.

- Type 2: Filtering is based on the RowFilter only.
  - If the RowFilter is equal to '*' then we scan the entire heap file as the star indicates that we have to scan all the files. The MapScan method calls the FileScanMap method.
  - If the rowFilter is given as a value/range, we use the processFilter method to extract the low and high keys from the range input. If range, we set the low and high keys, else we set the low and high to the same value.
- Type 3: The same operations are done as Type 2, but performed with the ColumnFilter instead.
- Type 4: Both the Row and ColumnFilter are considered together.
  - If both the row and columnFilter are specified as range values i.e columnFilterSize == 2 && rowFilterSize == 2, getKeyFilterForIndexType method is used to generate the condition expression after extracting the low and high key values based on both columnFilter and rowFilter range values delimited by "%".
    Ex: (index 0 represents the low value and 1 represents the high value)
    Low key:
    "columnKeyFilter.get(0).operand2.string + "%" + rowKeyFilter.get(0).operand2.string "
    High key:"columnKeyFilter.get(1).operand2.string + "%" + rowKeyFilter.get(1).operand2.string"
  - If RowFilter is a range value and the ColumnFilter is a single value, i.e columnFilterSize == 1 && rowFilterSize == 2, getKeyFilterForIndexType method is used to generate the condition expression with low key of rowFilter as low value along with columnFilter value delimited by "%" and high_key as rowFilter high value along with columnFilter value delimited by "%".

- If ColumnFilter is a range value and the RowFilter is a single value, i.e columnFilterSize == 2 && rowFilterSize == 1 getKeyFilterForIndexType method is used to generate the condition expression with low_key of columnFilter as low value along with rowFilter value delimited by "%" and high_key as columnFilter high value along with rowFilter value delimited by "%".
- If both rowFilter and columnFilter are fixed values, i.e columnFilterSize == 1 && rowFilterSize == 1, getKeyFilterForIndexType method is used to generate the condition expression with each of low_key and high_key set to rowFilter and columnFilter delimited by "%".
- If the columnFilter is * and rowFilter is fixed/range/*, then we do an entire heap scan.
- If the rowFilter is * and columnFilter is fixed value/range, we do the filtering based on the columnFilter.

- Type 5: We perform the same operations as above but using RowFilter and ValueFilter.

Task 2: Modifying the heap package

HeapFile:

Any table within the minibase environment is stored as a heapfile in the backend. In this particular implementation, one heapfile corresponds to one big table instance. Similar to a Tuple, within the HeapFile class, various functions are modified to support "Map" insertions instead of "Tuple" in the minibase environment. Since it is a heapfile, all the maps are maintained in an unordered fashion within the file.

The modifications in the HeapFile package are describes as under:

1. `MID insertRecordMap(byte[] recPtr)`: This function originally accepted a byte array as an input parameter, executed the insertRecord parameter of the HFPage and returned the corresponding RID. However, after the modification, this function accepts the byte array and inserts it into a heap file page and return the "MID" corresponding to the map.
2. `boolean deleteRecordMap(MID mid)`: Similar to the insertRecord, the deleteRecordMap is updated to support deletion of Maps from heap file pages. It takes the input MID to be deleted, locates the MID, deletes it and returns a boolean status.
3. `boolean updateRecordMap(MID mid, Map newmap)`: This function takes the input MID of the Map to be updated and a new Map with which it is to be updated. Once it locates the existing map in the heap file by the MID, it uses the mapCopy function from the Map class to replace the content of the maps.

4. `Map getRecordMap(MID mid)`: Using this function, the user can get the Map object corresponding to the existing MID.
5. `int getRecCntMap()`: This function provides the total number of Maps in the heapfile.
6. void deleteFileMap(): To delete any existing heapfile, this method is executed. It not only deletes the file, but also traverses the file directory header to delete the entry of the file from the directory.
7. `void openScanMap()`: To open and initialize a scan for traversing the maps in a heap file.

All the primary functions in this file are refactored to a new name where "Map" is appended at the end which indicates that it is a new functionality to support Map operations.

Similarly in the HFPage, all the supporting functions that assist with the insertion/updation/deletion of Tuples from the heapfile pages are modified to support Maps instead of Tuples. All the functions now use Maps or MIDs to locate Maps and work with them.

Also, various methods in the Scan class are updated to support different iterating operations like get_next, peekNext, mvNext etc. for a Scan object initiated on a heapfile that stores Maps.

Additionally, the RID Class is simply updated to MID which is referenced in multiple functionalities under the heap package. Since RID is nowhere referenced in the code, this is easily executed.

Sorting: After filtering, we check the orderType parameter and sort the values based on the given input. The filtered output from the previous step is given as input to this step. The orderType is of type int with the input integer performing the following operations.

1, then results are first ordered in row label, then column label, then time stamp
2, then results are first ordered in column label, then row label, then time stamp
3, then results are first ordered in row label, then time stamp
4, then results are first ordered in column label, then time stamp
5, then results are ordered in time stamp

<u>BigT - Version</u>

This Class is created to maintain the mapVersions of the indexes. It will strictly contain three indexes. The oldest one will be removed from the list. It has been implemented in the BatchInsert Class.

Task 3: Modifying the iterator

A utility class called MapUtils has been developed to handle certain operations on Map. It is very similar to the TupleUtils class and like other functions,this class has been created by modifying the TupleUtils class. It has been added to the iterator package and provides several static methods. These methods include

1. `int CompareMapWithMap(Map m1, Map m2, int mapfldno)`: This method compares the values of two maps m1 and m2 at a specified field number;
2. `boolean Equals(Map m1, Map m2)`: This method checks which checks whether all fields of two maps m1 and m2 are identical;

Task 4: Create bigDB class

The disk manager is one of the most integral parts of the DBMS, disk manager is responsible for allocating and deallocating pages, reading and writing of pages to and fro from the disk. The BigDB class is the Minibase implementation of a disk manager, it implements all the functionalities of a classic disk manager. These functionalities include reading/writing pages, allocating and deallocating pages, pinning and unpinning pages, opening and closing the database, deleting pages, creating pages etc.

Task 5: Count Read and Write Operations

The main objective of this task was to implement a counter that increments every time a disk access operation is performed. The objective was achieved by making changes to the disk manager package.

More specifically changes were made to the pcounter.java file within the diskmgr package. We declared two variables called the “rcounter” and “wcounter” which track the read and write operations respectively. An initialize () function is used to set the initial values to 0. Additionally, readIncrement() and writeIncrement() functions are implemented that increment “rcounter” and “wcounter” respectively. These functions are called within read_page() and write_page() that are enclosed within bigDB.java file, these functions are called every time a read or write operation is performed. Lastly, these counters are printed, every time batch insert or query implementations are invoked.

Task 6: Implement batchinsert program

The command line interface for batch insertion is as follows:

batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF

Let us breakdown the functionality of each identifier in the command

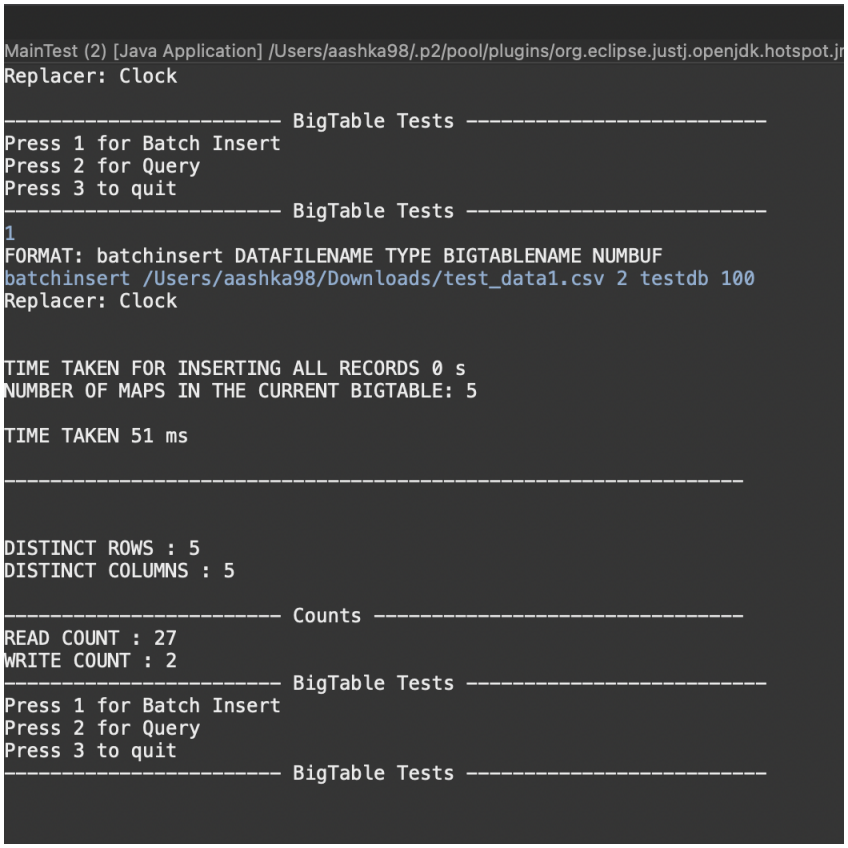
batchinsert: This acts as an identifier, to allow the interpreter to recognise the request for a batch insert operation.

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

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

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

Example:
batchinsert /Users/aashka98/Downloads/test_data1.csv 2 flop 100



Task 7: Query

The Utils package of the command line also demonstrates features for database querying. The interface for the query command is as follows:

query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF

USER INTERFACE

A separate class called MainTest has been created in the tests package. This class hold the main method. On execution, a menu pops up with 3 options as under.

```
---------- BigTable Tests ----------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
---------- BigTable Tests ----------
```

Option 1: Allows the user to execute the batch insert functionality. If the user enters 1, the class internally executes a batchinsert function, which splits the batch insert command into different segments and passes each segments individually. The file path that the user mentions is parsed and each line is read one after the other. For all the lines in the file, insertMap is executed individually.

Option 2: Allows the user to query the data that is just inserted. The user can also interact with the database by mentioning different filter conditions and also mentioning different orderTypes on which the data will be sorted.

Querying
The command line interface expects the query in the following format:

query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF

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

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

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

1 - No indexing
2 - Index row labels
3 - Index column labels
4 - Index column label and row label (combined key)
5 - One btree to index row label and value (combined key)

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

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

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

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

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

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

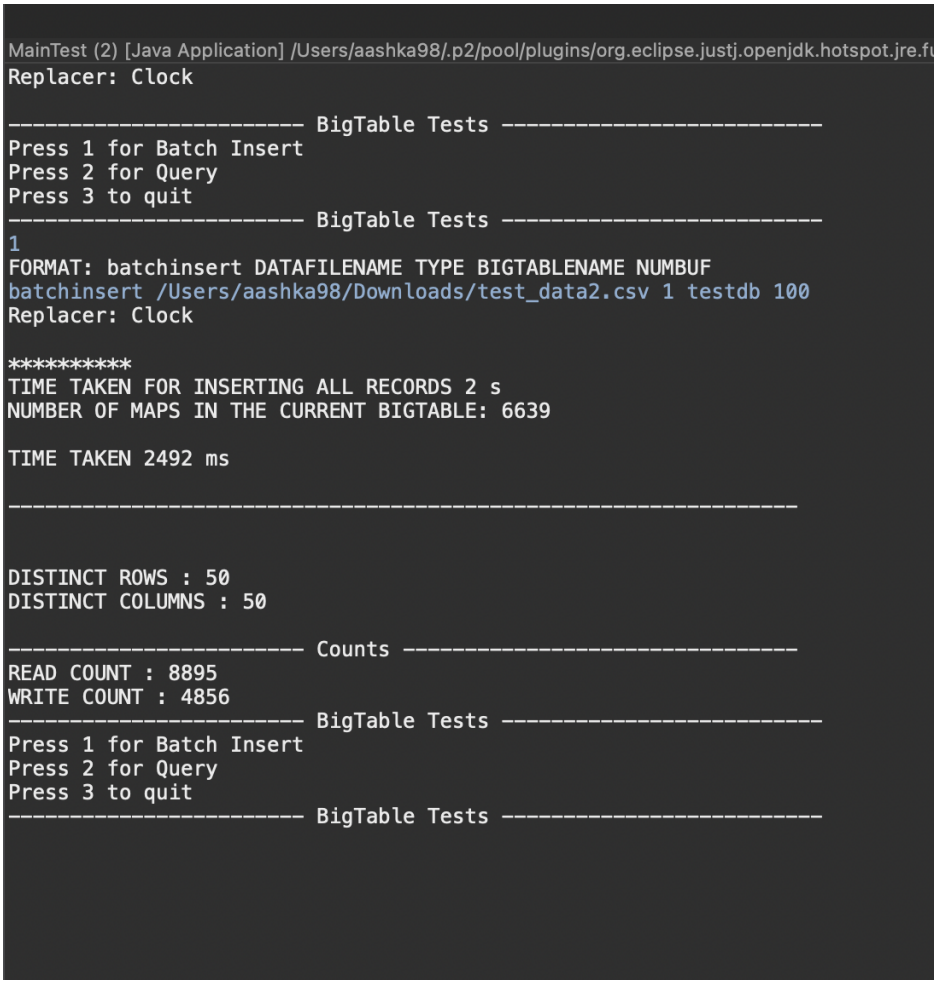
*Minibase will use at most NUMBUF buffer pages to run the query (see the Class BufMgr).

Example of a Query:
query testdb 2 1 * * * 100

OUTPUT & ANALYSIS

Batch Insert:

1. Evaluating Number of Reads and Writes when index type is 1 but number of buffers are different.



.

Number of Buffers: 100
Read Count: 8895
Write Count: 4856

```
----------------------- BigTable Tests -------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
----------------------- BigTable Tests -------------------------
1
FORMAT: batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
batchinsert /Users/aashka98/Downloads/test_data2.csv 1 testdb2 500
Replacer: Clock

**********
TIME TAKEN FOR INSERTING ALL RECORDS 2 s
NUMBER OF MAPS IN THE CURRENT BIGTABLE: 6639

TIME TAKEN 2483 ms

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



DISTINCT ROWS : 50
DISTINCT COLUMNS : 50

----------------------- Counts ---------------------------------
READ COUNT : 2982
WRITE COUNT : 2265
----------------------- BigTable Tests -------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
----------------------- BigTable Tests -------------------------
```

Number of Buffers: 500
Read Count: 2982
Write Count: 2265

As seen from the observation, number of buffers can significantly impact the read and write operations. If the number of buffers is less, the buffer manager would have to significantly read and write pages again and again from the DiskManager.

2. Understanding the relationship between number of buffers and read/write counts.

3. Evaluating Number of Reads and Writes when index type is 2.

```
------------------------ BigTable Tests --------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests --------------------------
1
FORMAT: batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
batchinsert /Users/aashka98/Downloads/test_data2.csv 2 batch1 100
Replacer: Clock

**********
TIME TAKEN FOR INSERTING ALL RECORDS 2 s
NUMBER OF MAPS IN THE CURRENT BIGTABLE: 7167

TIME TAKEN 2886 ms

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


DISTINCT ROWS : 50
DISTINCT COLUMNS : 50

------------------------ Counts --------------------------------
READ COUNT : 14868
WRITE COUNT : 9543
------------------------ BigTable Tests --------------------------
```

4. Evaluating Number of Reads and Writes when index type is 3.

```
------------------------ BigTable Tests --------------------------
1
FORMAT: batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
batchinsert /Users/aashka98/Downloads/test_data2.csv 3 batch3 100
Replacer: Clock

**********
TIME TAKEN FOR INSERTING ALL RECORDS 2 s
NUMBER OF MAPS IN THE CURRENT BIGTABLE: 7167

TIME TAKEN 2484 ms

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


DISTINCT ROWS : 50
DISTINCT COLUMNS : 50

------------------------ Counts --------------------------------
READ COUNT : 14858
WRITE COUNT : 9548
------------------------ BigTable Tests --------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests --------------------------
```

Looking at the read and write counts for type2 and 3, it is higher than type1. The increased

number of reads and writes is typically due to the creation and updation of indexes which is an added overhead to the normal insertion operation. These read and write counts would remain similar for index types 4 and 5 because only one index is being created in both the cases. Had an another index be included, the number of writes would significantly increase.

```
------------------------ BigTable Tests --------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests --------------------------
2
FORMAT: query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query testdb 2 1 * * * 100
Replacer: Clock

[Denmark Magpie 1 ] -> 61340
[Italy Peafowl 5 ] -> 70960
[Samoa Pinniped 3 ] -> 10163
[Sweden Moose 2 ] -> 43215
[Taiwan Cheetah 4 ] -> 71096
TIME TAKEN 19 ms
RECORD COUNT : 5

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


DISTINCT ROWS : 5
DISTINCT COLUMNS : 5

------------------------ Counts --------------------------------
READ COUNT : 20
WRITE COUNT : 0
------------------------ BigTable Tests --------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests --------------------------
```

Query:

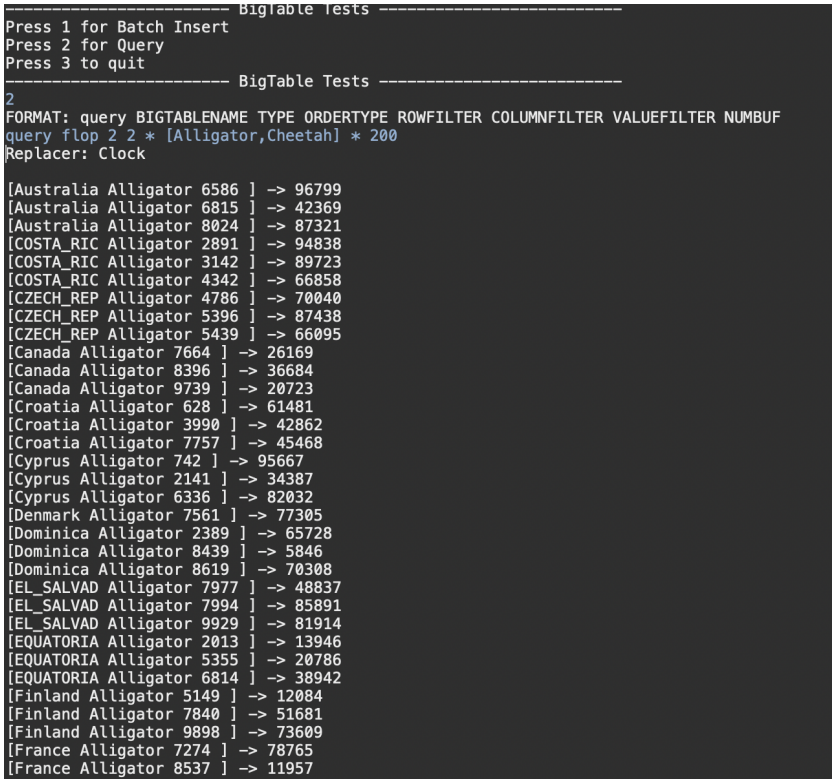
1. Querying the data with one row filter.

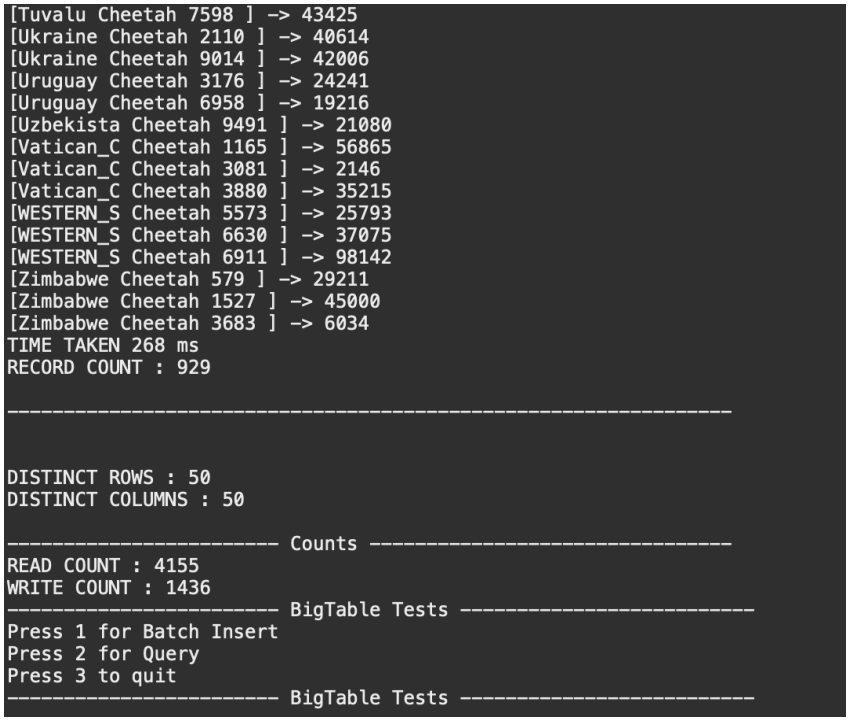
```
----------------------- BigTable Tests -------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
----------------------- BigTable Tests -------------------------
2
FORMAT: query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query flop 1 1 Taiwan * * 100
Replacer: Clock

[Taiwan Alligator 5197 ] -> 91032
[Taiwan Alligator 6516 ] -> 93136
[Taiwan Alligator 7633 ] -> 76127
[Taiwan Badger 5544 ] -> 8983
[Taiwan Badger 7778 ] -> 26782
[Taiwan Badger 9675 ] -> 3021
[Taiwan Beaver 311 ] -> 51974
[Taiwan Beaver 4208 ] -> 83925
[Taiwan Beaver 6129 ] -> 4881
[Taiwan Bison 5045 ] -> 25955
[Taiwan Bison 5336 ] -> 26426
[Taiwan Bison 7194 ] -> 34364
[Taiwan Black_pan 3854 ] -> 61408
[Taiwan Black_pan 5134 ] -> 28274
[Taiwan Black_pan 8798 ] -> 42477
[Taiwan Camel 2452 ] -> 30513
[Taiwan Camel 2670 ] -> 12261
[Taiwan Camel 3236 ] -> 25326
[Taiwan Cheetah 4 ] -> 71096
[Taiwan Cheetah 1645 ] -> 36935
[Taiwan Cheetah 7046 ] -> 63040
[Taiwan Cobra 3810 ] -> 47033
[Taiwan Cobra 5486 ] -> 48765
[Taiwan Cobra 8469 ] -> 59444
[Taiwan Columbida 5741 ] -> 37601
[Taiwan Columbida 8306 ] -> 18433
[Taiwan Columbida 9922 ] -> 9434
[Taiwan Coyote 6386 ] -> 11871
[Taiwan Coyote 6393 ] -> 39791
[Taiwan Coyote 8542 ] -> 18833
[Taiwan Crow 4403 ] -> 94439
[Taiwan Crow 5130 ] -> 62969
[Taiwan Crow 6900 ] -> 94954
[Taiwan Dolphin 5188 ] -> 82599
```

```
[Taiwan Rhinocero 5626 ] -> 40715
[Taiwan Sheep 5650 ] -> 10717
[Taiwan Sheep 8251 ] -> 14800
[Taiwan Sheep 9379 ] -> 19002
[Taiwan Skunk 3643 ] -> 64376
[Taiwan Skunk 9329 ] -> 46200
[Taiwan Skunk 9329 ] -> 46200
[Taiwan Swallow 5470 ] -> 83588
[Taiwan Swallow 6622 ] -> 20766
[Taiwan Swallow 8266 ] -> 59910
[Taiwan Wren 6533 ] -> 28710
[Taiwan Wren 8324 ] -> 93464
[Taiwan Wren 8670 ] -> 43267
[Taiwan Zebra 3417 ] -> 16948
[Taiwan Zebra 5311 ] -> 34506
[Taiwan Zebra 7262 ] -> 13753
TIME TAKEN 116 ms
RECORD COUNT : 149

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


DISTINCT ROWS : 50
DISTINCT COLUMNS : 50

------------------------ Counts --------------------------------
READ COUNT : 4493
WRITE COUNT : 1362
------------------------ BigTable Tests --------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests --------------------------
```

As seen in the output, the filter was successfully applied and only those records where columnLabel was “Taiwan” were displayed. Additionally, the order type 1 is also succesfully applied since first the rowLabel is sorted, followed by columnLabel, followed by timestamp and value.

2. Querying the data with range row filter.

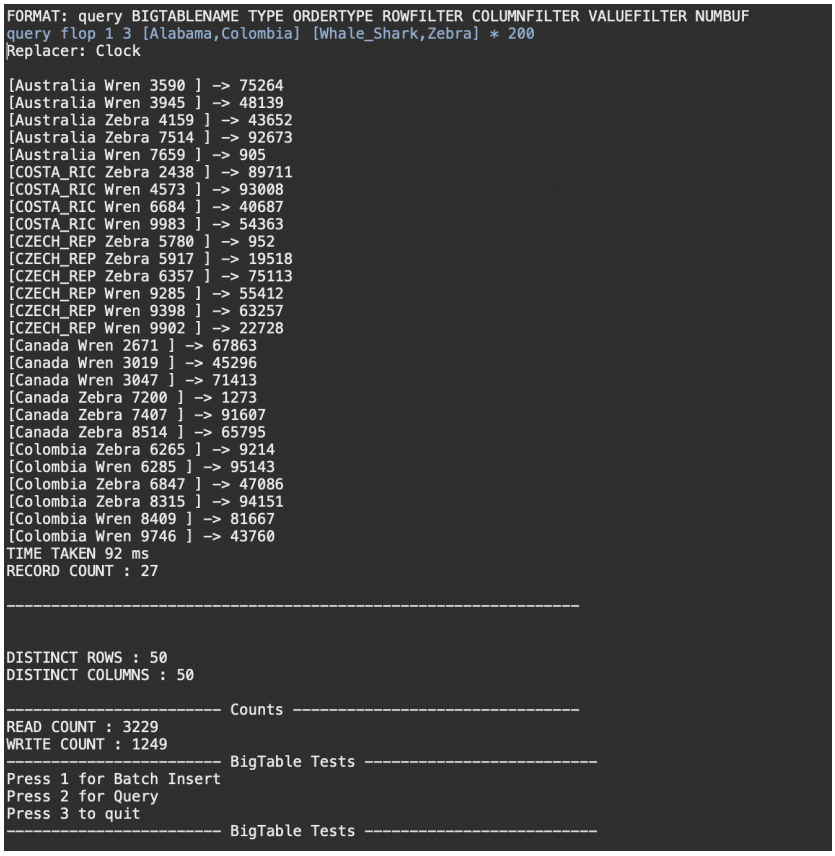




As seen in the output, a query was applied to fetch only maps where column label is

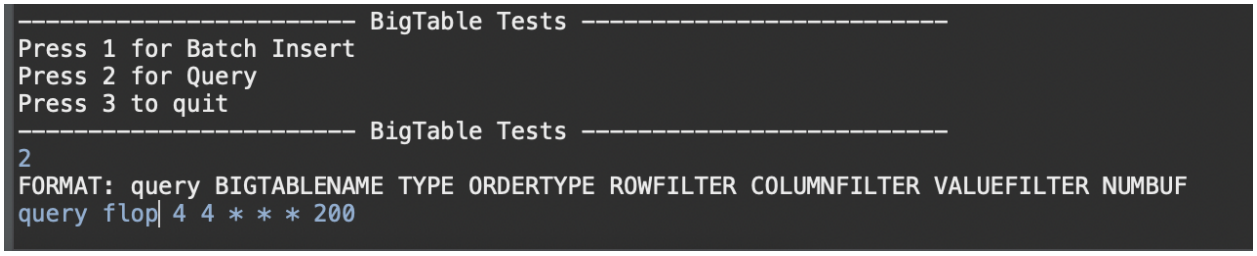
between Alligator and Cheetah. Additionally, the order type 2 indicates that the output is sorted byt columnLabel, followed by rowLabel followed by timestamp. This output is also successfully displayed in the image.

3. Querying the data with range filters in both rows and columns.



As seen in the output, a query was applied to fetch only maps where row label is between Alabama and Columbia and column label is between Whale_shark and Zebra. Additionally, the order type 3 indicates that the output is sorted by rowLabel, followed by timestamp. As seen in the output, the range filters are correctly applied on data and the data is also sorted on rowLabel followed by timestamp which is the expected behaviour of orderType 3.

4. Querying the data with all data.



```
[Finland Zebra 9144 ] -> 9359
[Nigeria Zebra 9171 ] -> 13745
[Tunisia Zebra 9197 ] -> 50460
[Slovenia Zebra 9220 ] -> 2717
[Cyprus Zebra 9241 ] -> 60720
[Tuvalu Zebra 9268 ] -> 46880
[Samoa Zebra 9562 ] -> 78557
[Ukraine Zebra 9579 ] -> 49004
[Italy Zebra 9601 ] -> 5085
[Netherlan Zebra 9636 ] -> 13672
[Samoa Zebra 9712 ] -> 75809
[Italy Zebra 9728 ] -> 62343
[Tuvalu Zebra 9826 ] -> 86772
[Kiribati Zebra 9903 ] -> 26986
[Nauru Zebra 9917 ] -> 40150
TIME TAKEN 1131 ms
RECORD COUNT : 6639

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


DISTINCT ROWS : 50
DISTINCT COLUMNS : 50

------------------------ Counts --------------------------------
READ COUNT : 4715
WRITE COUNT : 1968
------------------------ BigTable Tests --------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests --------------------------
```

When the query filters for rowLabel, columnLabel and value is * * *, it implies that there is no filter and all the data is to be considered. This query is paired with orderType 4 which indicates that the the output is to be sorted on columnLabel followed by timestamp. As seen in the output, the data is sorted on column followed by timestamp.

5. Querying the data with individual filter on column and row and an individual filter on value.

```
------------------------ BigTable Tests ---------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests ---------------------------
2
FORMAT: query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query flop 5 5 Denmark Magpie * 200
Replacer: Clock

[Denmark Magpie 1 ] -> 61340
TIME TAKEN 0 s
RECORD COUNT : 1
READ COUNT : 18
WRITE COUNT : 0
------------------------ BigTable Tests ---------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests ---------------------------
```

This is a simpler filter and filters only those maps whose rowLabel and columnLabel matches exactly to the filter in this case “Denmark”, “MagPie”.

```
------------------------ BigTable Tests ---------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests ---------------------------
2
FORMAT: query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query flop 5 5 * * 61340 200
Replacer: Clock

[Denmark Magpie 1 ] -> 61340
TIME TAKEN 0 s
RECORD COUNT : 1
READ COUNT : 16
WRITE COUNT : 0
------------------------ BigTable Tests ---------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests ---------------------------
```

This is a simpler filter and filters only those maps whose value matches the value filter as shown in the image.

6. Querying the data with ordering on timestamp.

```
[TRINIDAD_ Cobra 9971 ] -> 31934
[Tunisia Jackal 9972 ] -> 96911
[NEW_ZEALA Magpie 9973 ] -> 22944
[Netherlan Sheep 9974 ] -> 91778
[TRINIDAD_ Columbida 9975 ] -> 52716
[Grenada Black_pan 9976 ] -> 90888
[Denmark Sheep 9977 ] -> 47706
[NEW_ZEALA Beaver 9978 ] -> 81134
[Singapore Partridge 9979 ] -> 39319
[Morocco Mallard 9980 ] -> 29699
[COSTA_RIC Columbida 9981 ] -> 88082
[Tonga Nightinga 9982 ] -> 62174
[COSTA_RIC Wren 9983 ] -> 54363
[Sweden Quail 9984 ] -> 50596
[Finland Fish 9985 ] -> 9425
[Grenada Hyena 9986 ] -> 14965
[Samoa Flamingo 9987 ] -> 24726
[Tanzania Hornet 9988 ] -> 81606
[Kiribati Manatee 9989 ] -> 35677
[Vatican_C Partridge 9990 ] -> 35377
[CZECH_REP Flamingo 9991 ] -> 30668
[Colombia Crow 9992 ] -> 25175
[Croatia Wren 9993 ] -> 22197
[Sweden Gazelle 9994 ] -> 78703
[Spain Partridge 9995 ] -> 39297
[Uzbekista Donkey 9996 ] -> 10922
[France Hamster 9997 ] -> 73550
[Nigeria Alligator 9998 ] -> 78325
[Singapore Fish 9999 ] -> 22644
[Australia Giraffe 10000 ] -> 58338
TIME TAKEN 954 ms
RECORD COUNT : 6639

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


DISTINCT ROWS : 50
DISTINCT COLUMNS : 50

------------------------ Counts --------------------------------
READ COUNT : 5073
WRITE COUNT : 2000
------------------------ BigTable Tests -------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 to quit
------------------------ BigTable Tests -------------------------
```

This query is only for testing the orderType 5 where all the output is filtered on the timestamp.

SYSTEM REQUIREMENTS

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

EXECUTION

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

<PROJECT_DIRECTORY>/minjava/javaminibase/src

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

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

3. The following command executes the main program -

<JDKPATH>/bin/java tests.MainTest

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

1 for Batch Insert
2 for Query
3 to Quit

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

ENVIRONMENTS

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

LIBRARIES

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

RELATED WORK

The paper titled “Bigtable: A Distributed Storage System for Structured Data” introduces the concept of big tables [6], and walks us through a distributed storage system, capable of handling petabytes of data, which has reasonable flexibility and scalability. Big tables do not offer relational models; rather it emphasizes dynamic control over data layouts and

formats. Big tables allow clients of the database to have certain decision power over the location of their stored data, and to improve data access and manipulations, indexes are produced with respect to column and row labels.

CONCLUSIONS

Through this project, we transitioned a relational DBMS (Java Minibase) into a bigtable like DBMS, by changing the tuple based implementations into that of map based. This editing of the code base gave us confidence in the concepts, and opened the door to a deeper level of understanding. Furthermore, we implemented different combinations of indexes that allowed for an optimized query execution. We augmented the queries with other arguments such as Index type, buffer Number, order type that allowed us to experiment and analyze how smarter indexing and buffer choices can help improve performance of DBMS. Making granular changes to the code base, and then playing around with the query system has made us thoroughly familiar with concepts such as big tables, indexing and batch queries etc, and has elevated our understanding of the DBMS domain.

BIBLIOGRAPHY

[1] https://www.scylladb.com/glossary/wide-column-database/
[2] https://www.geeksforgeeks.org/difference-between-b-tree-and-b-tree/
[3] https://en.wikipedia.org/wiki/Bigtable
[4] Ramakrishnan, Raghu., and Johannes. Gehrke. *Database Management Systems* .3rd ed. Boston: McGraw-Hill, 2003.
https://byjus.com/gate/indexing-in-dbms-notes/#:~:text=Indexing%20in%20DBMS-,What%20is%20Indexing%20in%20DBMS%3F,present%20in%20a%20database%20table.
[5] https://research.cs.wisc.edu/coral/minibase/minibase.html
[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* (TOCS), 26(2), 1-26.

APPENDIX

Aashka Bhavesh Parikh -
Performed transformation on BigT class to transition into map based extension. Worked on BigT.bigt class. Coded constructors for BigT.bigt class, implemented filter matching mechanism. Coded constructors and getter/setters for BigT.Map class. Performed Debugging, testing, versioning and vetting of the code base.

Kshitij Dhyani -
Worked on the implementation of command line Batch insert and query invocations. Helped in the implementation of various filter mechanisms in Stream class. Worked on code integration. Researched most about BigTables and bought a general understanding of their inner working mechanisms. Worked extensively on the report structure and organization.

Himali Hemant Gajare -
Performed a plethora of querying by changing bufferNumber and index type. Observed read-write trends. Made changes to iterator classes, to accommodate map type data, worked on map comparison algo. Contributed abundantly towards the report and testing and debugging of the code base.

Maneesha Guntur -
Worked together with Aashka to make tuple to map transition of BigT and related classes. Specialized on work with BigT.Stream class to perfect different types of access mechanisms of opening, closing, of stream objects and retrieval of maps. Performed debugging, testing. Contributed to the understanding of query, indexing and batch insertion operations.

Rigved Alankar -
Worked primarily on the disk manager package, helped in the implementation of different indexing methods. Made modifications to disk manager in order to accommodate and store read write operations. Governed and led the direction of the project and largely helped with code versioning, and integrated different code branches.