Phase #2: BigTable DBMSI using Java Minibase Modules

CSE 510: Database Management System Implementation
Arizona State University
Spring 2020

by

Group #3 Team Members:
Kanishk Bashyam
Sumukh Ashwin Kamath
Baani Khurana
Rakesh Ramesh
Sreenivasan Ramesh
Ganesh Ashok Yallankar

Abstract: This report reflects the implementation of a BigTable DBMS that uses Java Minibase modules for support. The extension includes a map construct that has four fixed fields including row label, column label, timestamp, and value in replacement for tuples. Additionally, there are three main classes for the big table package: bigt, map, and stream. Heap file package is also modified to support the map construct. Btree based indexing methods for the map are further implemented within the disk manager package. Lastly, batch insert and query methods are also accounted for. To analyze the performance of different indexing techniques and query types, the number of disk reads and writes are counted.

Keywords: Java, Minibase, Buffer Management, Disk Space Management, Heap Files, B-trees, Index, Joins, Sort, Sort-Merge Joins, Functionality, Makefile, Query, Linux, Ubuntu, Typescript, BigTable, Map, Batch Insert, Row Label, Column Label, Timestamp, Value, Stream, Read, Write, Counter, Clustering.

TABLE OF CONTENTS

INTRODUCTION

Goals

The overall goals of this phase include understanding and implementing BigTable DBMS functionalities. This entails developing different storage schemes for indexing and clustering. An additional goal is to allow for batch insertions and different query types. The last goal is to analyze the performance of the different mechanisms and gather insights to determine conclusions.

To achieve these goals, many modifications are made to the Java Minibase modules including heap package, disk manager, and iterator package, etc. By incorporating the “Map” BigTable concept as a substitute for the relational “Tuple” concept, the BigTable DBMS functionalities are successfully implemented. Additionally, performance is measured by counting the number of read and write disk accesses.

Terminology

Database: A database is a collection of “related” observations, a set of data, that is recorded and organized for efficient retrieval [1]. There are different types of data models that formally describe data storage mechanisms for a database such as physical models including data structures, logical models including relational, OO, and OR, and conceptual models such as UML, ER, and Extended ER. The tradeoffs include performance and the expressive power.

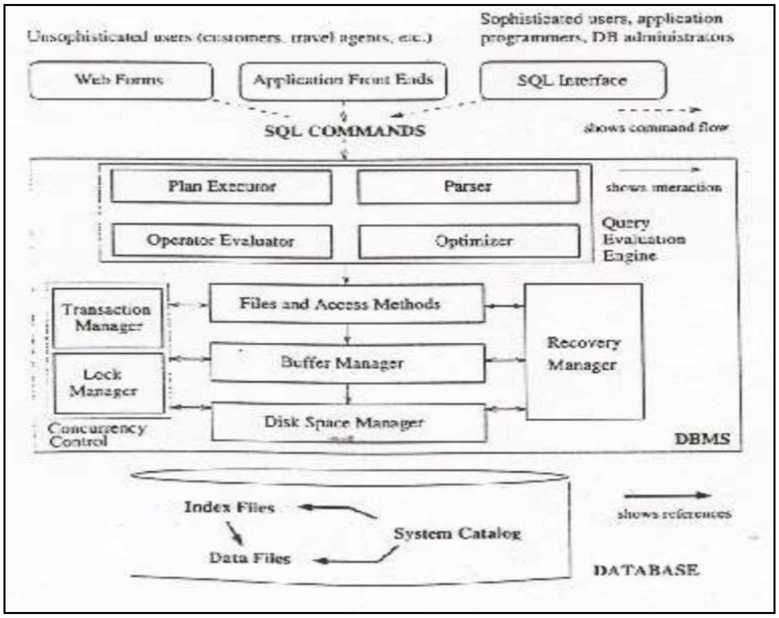


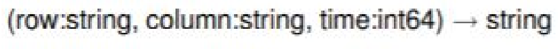
Figure 1. Architecture of a DBMS

Database Management System (DBMS): A software and hardware system for storage, retrieval, and manipulation of data [2]. The DBMS supports efficient search, queries, and automation. It also enforces constraints, enables concurrency and transactions of data, and maintains consistency and recovery. The architecture of a DBMS includes a System Catalog, Disk Manager, Buffer Manager, Recovery Manager, Concurrency Control, and Query Optimizer, etc.

Java Minibase: A relational DBMS that is traditionally row-store. It contains many functionalities described in a DBMS. These Java minibase modules serve as the foundation of a transformation into a big table DBMS using "Map" constructs instead of "Tuple" constructs.

Bigtable: Bigtable is a "distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers" [3]. It provides flexibility and high performance with sparsity, distribution, persistency, and multi-dimensionality.

Map: A construct with four fixed fields: row label (string), column label (string), timestamp (integer), and value (string). Each value in the map is an array of bytes.



(row:string, column:string, time:int64) → string

Figure 2. Declaration of a Map Concept

Disk Space Manager: This part of DBMS focuses on storage of pages within a database such as allocation and deallocation. It conducts read and write operations to and from the disk. The Disk Space Manager (DSM) preferably needs to be robust and durable. Overall, DSM is responsible for reading from the disk, writing to the disk, allocating pages within the database, and deallocating pages through the Database class and Page class.

Buffer Manager: The purpose of the buffer manager is to switch disk pages in and out of the main memory [1]. It writes to new pages, reads, and releases the pages to the operator. This is a critical feature and needs to be managed efficiently because there is a limited amount of room in the buffer.

File Manager: There are two main roles of a file manager: to organize the disk pages into a coherent file and to store rows into pages. The goal is to acquire and link disk pages and then organize records within the disk pages. There are two types of files: data files such as a heap file and index files.

Heap File: A type of data file where the records in the pages are unsorted [2].

Index File: A data structure that helps locate and access the data in an efficient manner [1].

B+ Tree: A tree structured index file where the tree height is balanced. The leaf pages contain data entries and are chained, while non-leaf pages have index entries [1]. B+ trees support equality and range searches. They are useful for dynamic data including insertions and deletions. An important criteria of

B+ trees is to maximize fanout to allow for more utilization and pruning [2]. Additionally, it may be costly to insert large records. Therefore, bulk loading is also supported.

Indexing: A mechanism to speed up and maximize performance of queries by minimizing disk accesses within a database. It utilizes selections on the search key fields for efficient retrieval.

Query: A query is like a search to retrieve specific data from a database. There are many different types of queries. In this project, there are various parameters that determine the type of query such as the index type, attribute by which records are ordered, a row filter, a column filter, and a value filter. Each filter can either be unspecified, a single value, or a range of values.

Assumptions

1. We are considering the Value field as of data type string (`java.lang.String`). We are not padding with 0's as padding with 0's works well with numbers but gives incorrect results with characters. Example: Alpha, Beta: Since Beta is only 4 characters long, padding Beta with a 0 (0Beta) makes it smaller than Alpha.
2. Every BIGTABLENAME has to be unique, and you cannot have the same BIGTABLENAME with different types. Batch Insertion can also be performed only once per BIGTABLENAME. We decided on this approach as the number of pages is fixed during creation and the DB will have a fixed size, and Minibase does not provide us to dynamically change the number of pages.
3. The row field, column field and value field are assumed to be of String data types (`java.lang.String`).
4. The TimeStamp field is assumed to be of Integer data type (int) and the values in the timestamp field are considered in seconds. Millisecond values need long support and we are currently assuming only values are only in seconds (int).
5. It is assumed that The Range filters are closely packed in the query, without any whitespace between the range values.
6. If the size of the map object exceeds the page size, an error is thrown.

SOLUTION

Implementation

Task 1: BigT Class

BigT.bigT

This class is mainly used for the creation and maintenance of the BigT table. This is a class that maintains all the relevant heap files and the index files required for the BigT table. It provides constructors to open or create the Big table. It also provides a **close()** method to close the operations of the BigT table. The constructor which takes just the name as input is used to open an existing Big Table and the constructor which takes both the name and type of the clustering and indexing strategy is used to create a new Big Table. There are 5 possible values for indexing types. They are:

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

InsertMap(): This is a method to Insert a new map into the Big table. This method takes care of maintaining only 3 recent versions for a particular row and column label. When a fourth map is being inserted, the map with the oldest timestamp is removed in the big table. Since we will be dealing with bulk inserting of records, checking for the versions of the map for every insert involves fetching a lot of records from the Heapfile and this becomes very inefficient and time consuming. In order to avoid this bottleneck, we maintain a Hashmap with the key being a string which is a concatenation of row and column values and the value being the list of MIDs with the 3 most recent timestamps for that key. So when a new map is being inserted, we simply check this Hashmap to determine the versions and delete the oldest version. This method also takes care of inserting the SearchKey which is determined based on the type of the index and the Map pointer to the map in the Heap file in the relevant btree index file, which is maintained by the Big Table.

There are three methods which are provided to return the counts of various maps in the Big table. getMapCnt() returns the total number of maps stored in the Big table. getRowCnt() returns the number of distinct rows in the Big table and getColumnCnt() returns the number of distinct columns stored in the Big table. These two methods leverage the use of Hashmap created in the InsertMap to efficiently retrieve the counts.

The openStream() is a method which is used to instantiate a Stream of maps. This stream of maps match the rowFilter, columnFilter and valueFilter that is passed as input to this method. The implementation of the Stream class is discussed in one of the following sections. The way the results should be ordered is determined by an orderType integer which is also passed to this method. The possible values for orderTypes are:

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

Certain metadata files are created to maintain metadata like the type of the indexing/clustering strategy being used for the Big table and the Hashmap which is used to insert a map to the Big Table.

BigT.Map

A map is similar to a Tuple, but we have 4 fixed fields - row, column, timestamp and a value. The Map class provides functionality to create a new map object. Each map object has a header, which provides the information on each field offset in the data structure. The Map object has 4 fields namely row, column, timestamp and value. The row, column and value fields are strings which are of variable length. The timestamp field is an integer of size 4 bytes. The starting position of these fields are stored in the header. The header consists of multiple fields beginning at the number of fields stored in the data structure followed by the starting position (offset) of the first element which is the row and then the offset for column and timestamp and value. The last position will contain a value which is the ending position of the record which will be the map offset of a new record which will be placed after this record. The header structure consists of short fields. We considered having a fixed length record structure, although it's easy to store and retrieve the records, we might be allocating more space than needed and also the entire string may not fit in the space allocated. We decided to use variable length records to fit in more records per page causing fewer page reads. Minibase sort doesn't support the variable length records. We may be modifying it in the future to support variable length records in the sort. So we had a variable size record which has a fixed structure per table. The map structure for each field has the maximum length set per column which is obtained during the batch insert. So the structure is variable length but fixed length per table.

The map has various methods to access the data and set data fields.

* `Map()` : Class constructor create a new map with the appropriate size.
* `Map(byte[] amap, int offset)`: Construct a map from a byte array.
* `Map(Map fromMap)`: Construct a map from another map through copy.
* `java.lang.String getRowLabel()`: Returns the row label.
*`java.lang.String getColumnLabel()`: Returns the column label.

* `int getTimeStamp()`: Returns the timestamp.

* `java.lang.String getValue()`: Returns the value.

* `Map setRowLabel(java.lang.String val)`: Set the row label.

* `Map setColumnLabel(java.lang.String val)`: Set the column label.

* `Map setTimeStamp(int val)`: Set the timestamp.

* `Map setValue(java.lang.String val)`: Set the value.

* `byte[] getMapByteArray()`: Copy the map to byte array out.

* `void print()`: Print out the map.

* `size()`: Get the length of the map

* `mapCopy(Map fromMap)`: Copy the given map

* `mapInit(byte[] amap, int offset)`: When you don't want to use the constructor.

* `mapSet(byte[] frommap, int offset)`: Set a map with given byte array and offset.

BigT.Stream

The Stream class in BigT is analogous to the Scan class of the heap package. It provides a way to "stream" through the maps in a bigtable. The stream class provides a getNext method which returns the next map element satisfying the filters and the orderType constraints. To do this we do the following.

First, depending on the index type and the filter, we set a class variable which tells if we will scan the entire heapfile or search the index files.

Type 1: We scan the entire heap file.

Type 2:

- If the rowFilter is *, then we scan the entire heap file. We could scan on the index file also, but the BTreeFile's new_scan method requires a lo_key and a hi_key to be passed. We currently are not storing the lowest and highest values in the metadata, and are scanning the entire file when the filter is *. The advantage of using an index file over the heap file is that the index file maintains the maps in a sorted order. We will be changing this in future implementations.
- If the rowFilter is a value or a range, we are using the BTreeFile's new_scan method to scan the index. If range, we set the lo and hi keys, else we set the low and hi to the same value.

Type 3: We perform the same as above but for the columnFilter instead of the rowFilter.

Type 4:

- If the rowFilter is * and columnFilter is *, then we scan the entire heap.
- If the rowFilter and columnFilter both are range values, We use BTreeFile's new_scan method with low_key and hi_key is based on both columnFilter and rowFilter range values delimited by "$".
- If the rowFilter is range and columnFilter is fixed value, We use BTreeFile's new_scan method with low_key based on rowFilter low value and columnFilter delimited by "$", high_key is based on rowFilter high value and columnFilter delimited by "$".
- If the rowFilter is range and columnFilter is *, we use BTreeFile's new_scan method with low_key based on rowFilter low value only, high_key is based on rowFilter high value.

- If the rowFilter is fixed value and columnFilter is range, We use BTreeFile's new_scan method with low_key based on rowFilter value and columnFilter low value delimited by "$", high_key is based on rowFilter value and columnFilter high delimited by "$".
- If the rowFilter is * and columnFilter is range, We use BTreeFile's new_scan method with low_key based on columnFilter low value, high_key is based on columnFilter high value.
- If the columnFilter is * and rowFilter is fixed, then we scan the entire heap.
- If the rowFilter is * and columnFilter is fixed value, We use BTreeFile's new_scan method with low_key and high_key both set to columnFilter.
- If the both rowFilter and columnFilter are fixed values, We use BTreeFile's new_scan method with low_key and high_key set to rowFilter and columnFilter delimited by "$".

Type 5: We perform the same above operations with rowFilter instead and columnFilter and ValueFilter instead of rowFilter respectively.

Filtering: Once we decide which file to scan, we then filter the data, and add it to a temporary heap file.

- Heapfile: If we are scanning the entire heap file and all the filters are * then we initialize the temporary heap file to the original heap file as it will have the entire data. Otherwise, we open a mapScan and iterate through the maps, and check each entry if it satisfies our filter conditions. If all the conditions are satisfied the data is added to a temporary heap file.
- Indexfile: If we are scanning the index file, we use the BTreeFile's get_next() method to get the next entry. Once we get the next entry, we check the record against the filter conditions, and if they are all satisfied the data is added to the temporary heap file.

Sorting: Once the data is filtered, we then check the orderType to decide how to sort the data. Based on the orderType, we set the field to be sorted in *sortField.* The above filtered Heapfile or the Index File is passed to the Sort object as iterator for sorting.

Sorting is based on OrderType:

1. Sorted by sort field 1 - row field, column field and timestamp value.
2. Sorted by sort field 2 - column field, row field and timestamp value.
3. Sorted by sort field 1 - row field and timestamp value.
4. Sorted by sort field 2 - column field and timestamp value.
5. Sorted by timestamp value.

Task 2: Modify Heap Package

Heap file is a type of file organization where the records are stored in unsorted order. In Minibase, the tuples are stored in Heap files. In BigT, we have modified the Heapfile.class and the other related classes in the Heap package to accommodate the storage of Maps in the heap files. The maps are not stored in any particular order in the heap file, they are stored into the empty slots as and when the maps are inserted.

Each map in the Heap file is associated uniquely by an identifier called Map Identifier, which is also known as the MID. Each map is uniquely identified by the page number and the slot number in which the map resides and we have created a class called MID.class to encapsulate this information. This class has getter and setter methods to get and set the page number and slot number, a constructor to instantiate the class and a method equals() to compare one MID with another.

Heapfile: The heap file consists of two types of pages: directory pages and data pages. Both directory pages and data pages are of the type HFPage(). The first directory page is the header to the entire database. Each record in the directory page points to a data page where the actual maps are stored. The directory pages are stored as a doubly linked list. The data pages have slots at the beginning and the maps are stored after the slots in the page and are referenced by the slot number. The existing heapfile class is modified to include the following methods: insertMap() which takes a byte[] array as input and inserts the map into an empty slot in the data page and returns the MID of the inserted Map, deleteMap() which takes the MID as input and deletes the map present in that particular slot and marks the slot as EMPTY_SLOT. This slot can then be used to insert maps in the future, getMap() which takes a MID as input and fetches and returns the map present at the particular slot number, getMapCnt() which returns the number of maps present in the Heap file, updateMap(), which takes an MID and a Map as input and updates the map pointed to by the MID and updates it with the input map. openMapScan(), which instantiates a MapScan class using the current heap file and returns an instance to that class.

The HFPage class is also suitably modified to provide methods to insert the map, return the map, get the first map in the HFPage and the next map to an existing map in the HFPage.

The MapScan: This is a new class which is provided through the openMapScan method of the Heapfile. This is a method to scan through all the maps in a given heapfile. The maps are retrieved in the order in which they are stored in the heapfile. This class provides a getNext() method to retrieve the next map stored in the Heapfile.

Task 3: Create Map Utils Class

We have created a utility class for handling the processing of certain operations on Map. This MapUtils class is added in the iterator package. These are some of the static methods that are provided in this class: CompareMapWithMap() takes two maps and a field number and compares the values of the two maps at the given field number, Equals() which takes two maps as input and returns if all the fields of the two maps are equal, CompareMapsOnOrderType() takes into consideration the OrderType to compare the two maps.

Task 4: Create bigDB Class

The disk space manager is the component of the system that takes care of the allocation and deallocation of pages among the different files of a database. It also performs reads and writes of pages to and from disk. In Minibase, a database is implemented as a single UNIX file[1]. The BigDB class is similar to DB.java in minibase, it acts like a disk manager, provides various functionalities like, Opening the database, closing the database, deleting the database, writing pages, allocating pages, reading pages, deallocating file pages, pinning pages, unpinning pages, and it has sub classes for creating header pages and directory pages. The DB class also provides a file naming service . This service is implemented using records consisting of file names and their header page ids. There are functions to insert/delete/access file entries[1].

Task 5: Count Reads and Writes

The goal of this task is to add functionality that can monitor disk accesses to help analyze the performance of different batch inserts and queries. In order to achieve this goal, the disk manager has been modified to track the number of reads and writes.

Within the **diskmgr** package, a **pcounter.java** class has been added with two class variables called "rcounter," which counts read accesses and "wcounter," which counts write accesses. Additionally, three methods have been created. Firstly, ***initialize()*** method which sets "rcounter" and "wcounter" both to zero. This method is called within the *batchInsert()* and *query()* methods under **Utils.java** class to initialize and reset the counters everytime a database is opened within a query or batch insert. Secondly, ***readIncrement()*** increments the "rcounter" variable and is called within the *read_page()* method under **bigDB.java** class to track the amount of reads. Thirdly, ***writeIncrement()*** increments the "wcounter" variable and is called within the *write_page()* method under **bigDB.java** class to track the amount of writes.

Finally, within batch insert and query implementations, the values for "rcounter" and "wcounter" are printed to the console for detection of the performance.

Task 6: Batch Insert

The **cmdline** package provides a Utils class that implements a CLI to perform Batch Insert and Querying tasks. To run batch insert, we first compile and run MiniTable, which gives us a CLI. In the CLI, we can run both batch insert and query commands. The batch interface command's interface is as follows:

```
batchinsert DATAFILEPATH TYPE BIGTABLENAME
```

Note:

- The absolute path to the datafile csv path should be passed.
- Every bigtable name is unique. You cannot have multiple bigtables with the same bigtable name.
- You cannot batch insert into a minitable a second time. We decided not to support this as we have fixed the size of the DB (num pages is fixed) while creating it .

A sample batch insert looks like:

```
miniTable>  batchinsert /Users/vasan/testdata.csv 1 type 1
/tmp/type.db
Replacer: Clock


========================================

map count: 28252
Distinct Rows = 101
Distinct Coloumns = 100

========================================

Reads : 67029748
Writes: 43184
NumBUFS: 400

========================================

Total execution time: 247.795
```

Implementation:
We create a new bigT object and then read each line of the csv and create a map obj with the data. We use bigT's insertMap to insert data into the database file. The insertMap function also takes care of versioning to maintain only the 3 most recent values. To do this, during the batch insert, we create a Hashmap that contains the $rowValue\$columnValue$ as the key and a List of 3 most recent MIDs as the value. When a map is inserted, we check this Hashmap to check if the 3 MIDs are already present. If present, we remove the oldest map and any indexes if present, and replace it with the new map and update the Hashmap.

Task 7: Query

The cmdline's Utils package also proves functionality for querying the database. The query command's interface is as follows:

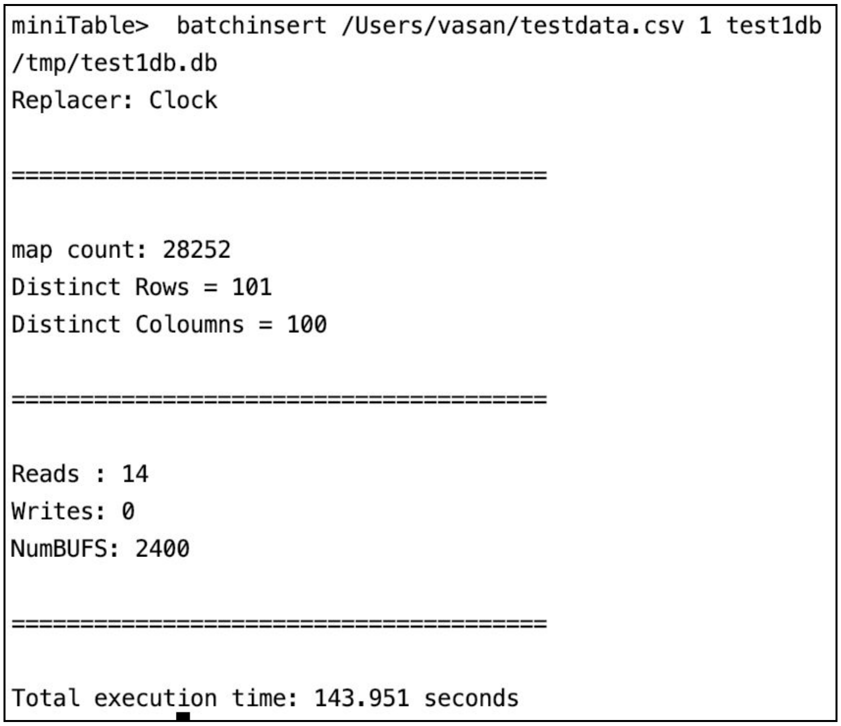
```
query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
```

Note: The bigtablename should already exist. Also if a range filter is used, no space should be given in between two values.

Output and Analysis

Batch Insert Tests

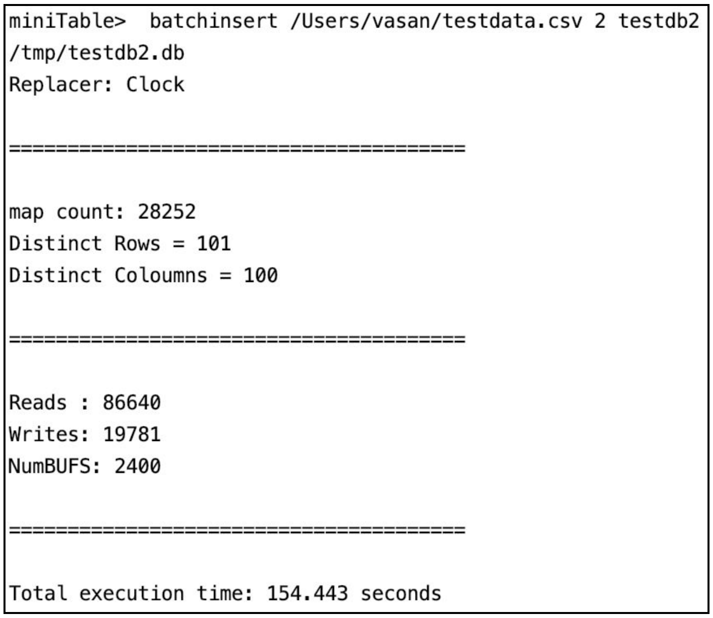
<u>Type Index 1</u>



```
miniTable>  batchinsert /Users/vasan/testdata.csv 1 test1db
/tmp/test1db.db
Replacer: Clock

=======================================

map count: 28252
Distinct Rows = 101
Distinct Coloumns = 100

=======================================

Reads : 14
Writes: 0
NumBUFS: 2400

=======================================

Total execution time: 143.951 seconds
```

We initially had fixed length records for 32 byte strings which gave us around 194 million reads. We modified it to set the field lengths based on the maximum field length given in the csv. This allowed us to fit more records into a page. We also use temporary metadata files, which reduces the number of reads significantly. Changing the NumBUFS to 2400 also helped us significantly reduce the reads and execution time.

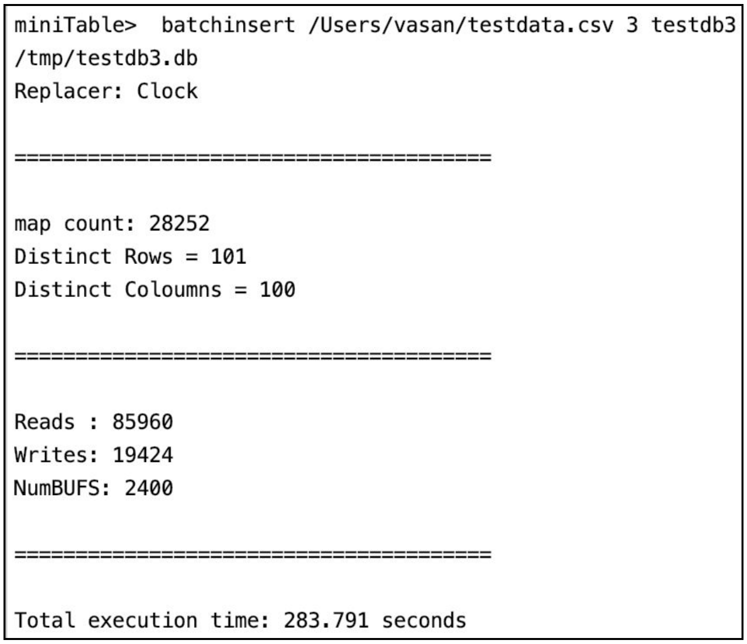
Type Index 2



```
miniTable>  batchinsert /Users/vasan/testdata.csv 2 testdb2
/tmp/testdb2.db
Replacer: Clock

=======================================

map count: 28252
Distinct Rows = 101
Distinct Coloumns = 100

=======================================

Reads : 86640
Writes: 19781
NumBUFS: 2400

=======================================

Total execution time: 154.443 seconds
```

The number of reads is higher compared to the previous batch insert due to the cost of constructing and updating the index file. This makes the reads and writes to increase along with the execution time.

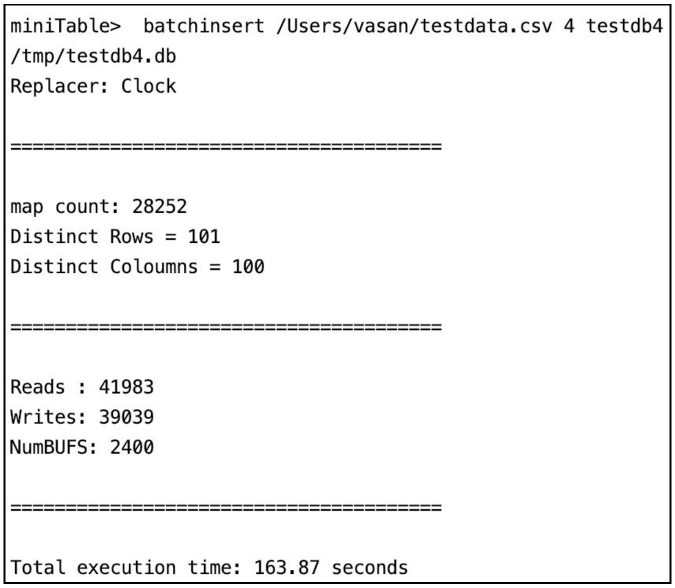
Type Index 3



```
miniTable>  batchinsert /Users/vasan/testdata.csv 3 testdb3
/tmp/testdb3.db
Replacer: Clock

=======================================

map count: 28252
Distinct Rows = 101
Distinct Coloumns = 100

=======================================

Reads : 85960
Writes: 19424
NumBUFS: 2400

=======================================

Total execution time: 283.791 seconds
```

The inserts here are similar to the one before, but the execution takes longer time. There are a lot of similar values in the column, because of which the index creation takes a longer time. Since our NumBUFS is high, the number of reads and writes are not much higher compared to type 2.

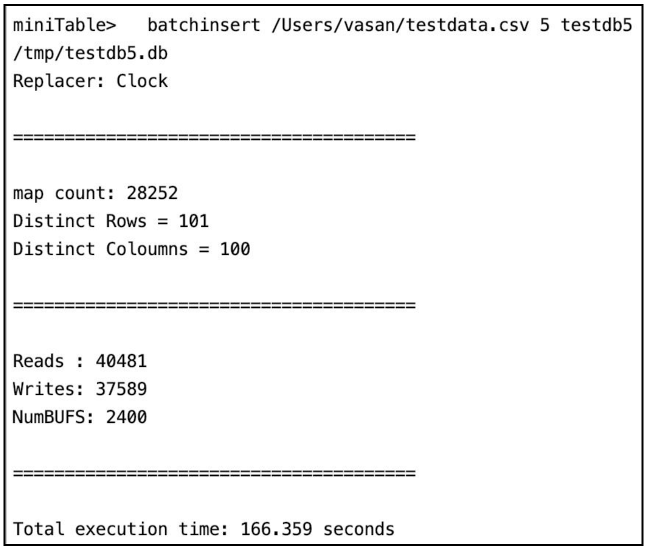
Type Index 4



```
miniTable>  batchinsert /Users/vasan/testdata.csv 4 testdb4
/tmp/testdb4.db
Replacer: Clock

=======================================

map count: 28252
Distinct Rows = 101
Distinct Coloumns = 100

=======================================

Reads : 41983
Writes: 39039
NumBUFS: 2400

=======================================

Total execution time: 163.87 seconds
```

In this case the number of writes are higher compared to the second and third cases. The increase in the number of writes is due to the creation and updation of indexes. We are creating two indexes, a composite index and a timestamp index, which increases the number of writes. The number of reads is lesser as the indexed values are more unique.

Type Index 5

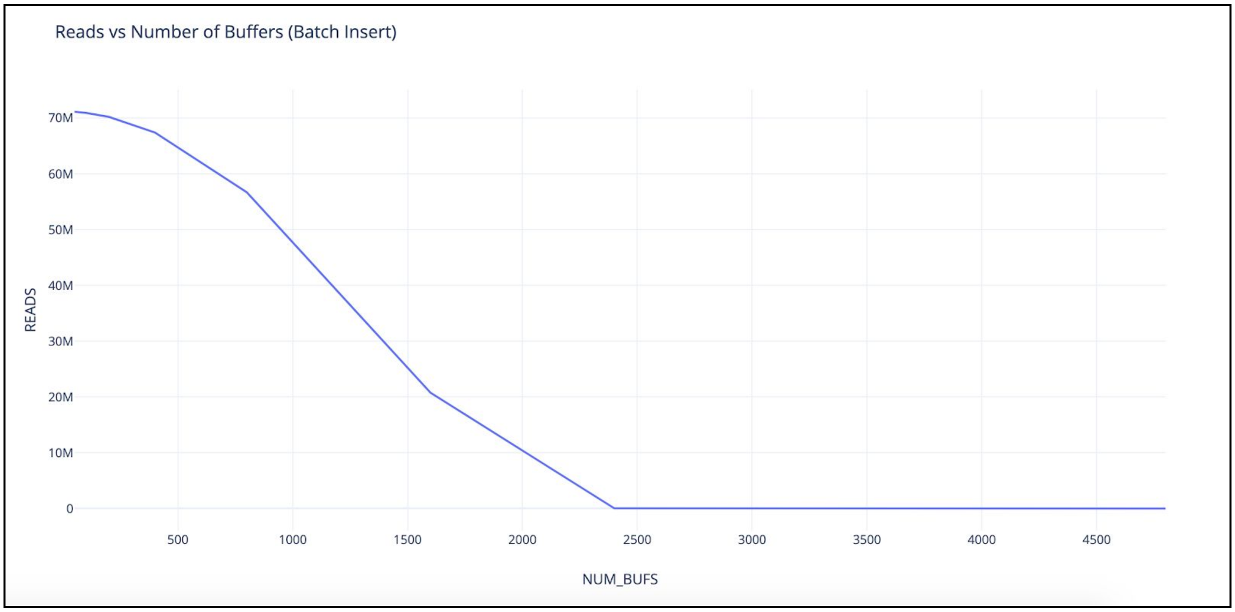


```
miniTable>   batchinsert /Users/vasan/testdata.csv 5 testdb5
/tmp/testdb5.db
Replacer: Clock

=======================================

map count: 28252
Distinct Rows = 101
Distinct Coloumns = 100

=======================================

Reads : 40481
Writes: 37589
NumBUFS: 2400

=======================================

Total execution time: 166.359 seconds
```

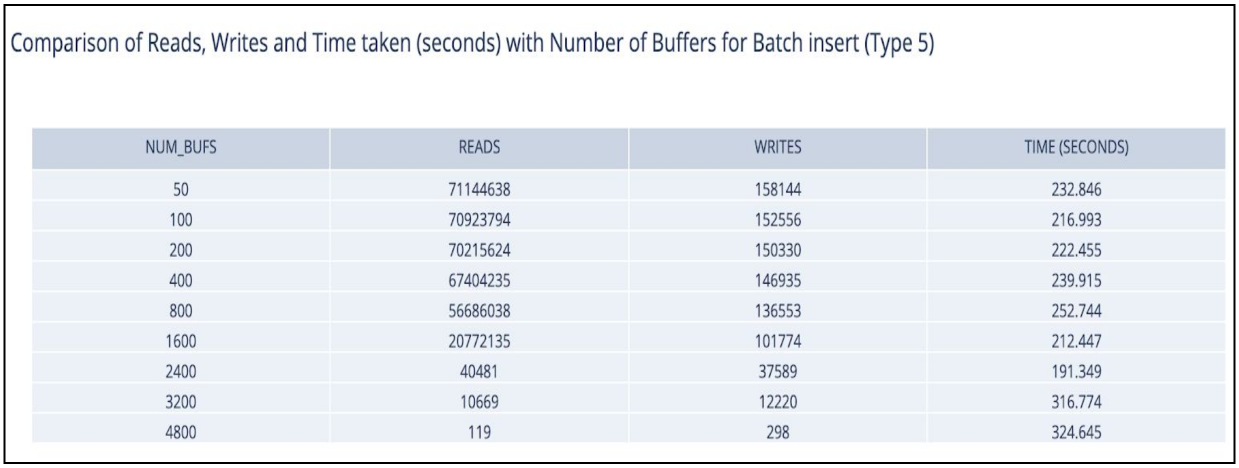
This indexing option is similar to the 4th index type, there are two indexes being created here, which increases the writes and execution time.

NUMBUF Tests

We tested the number of reads, writes and noted the execution times by changing the number of buffers and the above table shows the results.
With the increase in the number of buffers the cache hit ratio is high, Three graphs are plotted with these results, each graph with respect to the number of buffers.

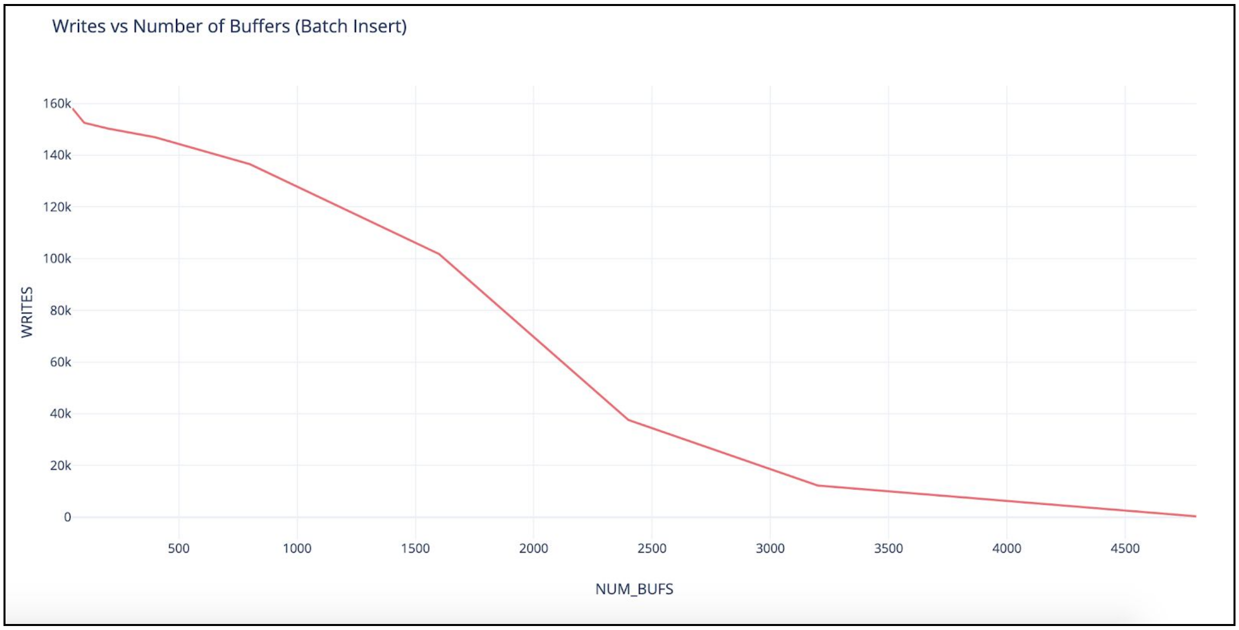


The above graph for the number of reads and buffers. With the increase in the number of buffers, The number of reads significantly decreases and then saturates. If there are high buffers, all the reads happen on the buffer and hence there are less page reads which is reflected in the graph.

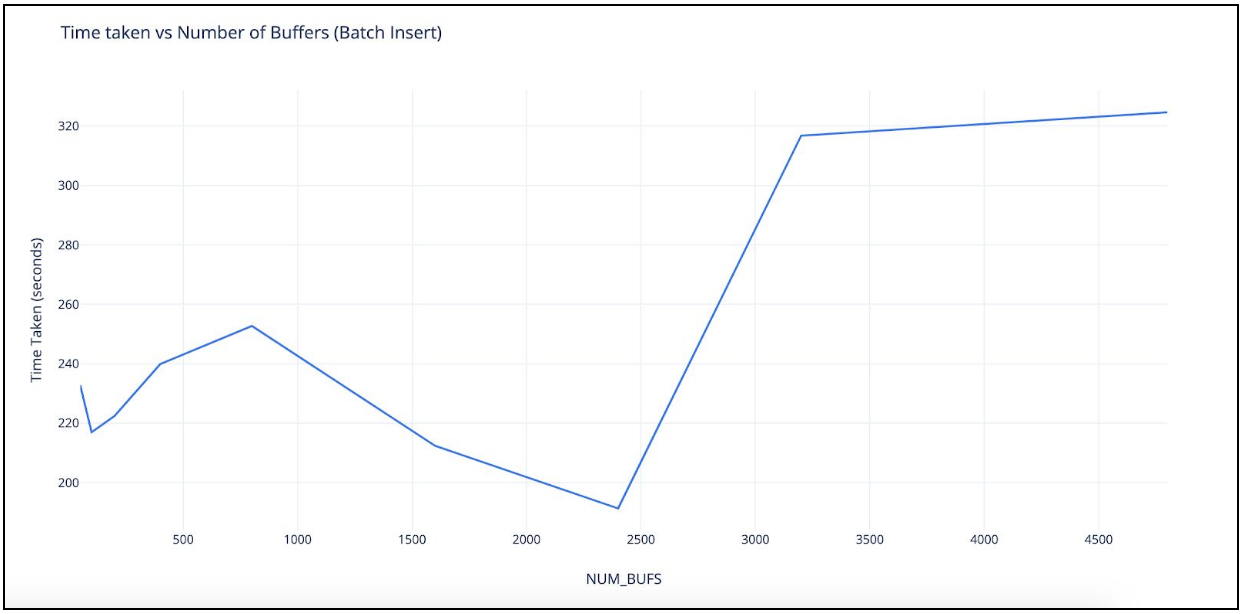


Comparison of Reads, Writes and Time taken (seconds) with Number of Buffers for Batch insert (Type 5)

NUM_BUFS	READS	WRITES	TIME (SECONDS)
50	71144638	158144	232.846
100	70923794	152556	216.993
200	70215624	150330	222.455
400	67404235	146935	239.915
800	56686038	136553	252.744
1600	20772135	101774	212.447
2400	40481	37589	191.349
3200	10669	12220	316.774
4800	119	298	324.645



The above graph shows the number of writes for varying buffer sizes, the number of page writes decreases. Since there are a higher number of buffers, the writes happen more on the buffers instead of pages and hence the number of writes decreases with the increase in buffer count.



The above graph shows the time taken for batch insertion of total size 50000 records. With the increase in the number of buffers, the time slightly increases and the time taken decreases rapidly until the number of buffers reaches 2400, this was where the number of reads in the data pages decreased very rapidly. The cache hit ratio has saturated at this point. After this point increasing the number of buffers increased the time taken to insert. The main reason for this is the number of dirty pages are very high at this point and there will be large overhead when flushing these to the disk and its much harder for buffer synchronization which is making the time to increase.

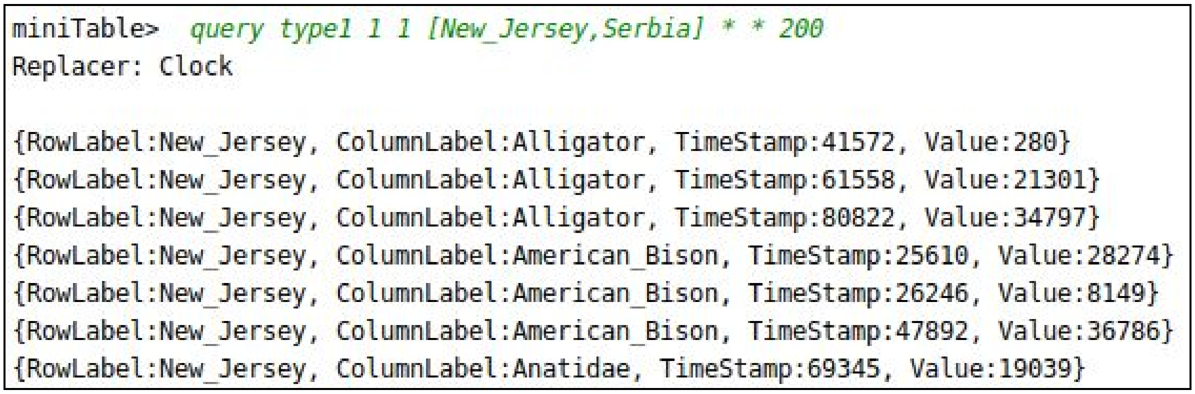
Query Tests

Type Index 1

Order Type 1: Row Label, then Column Label, then Timestamp

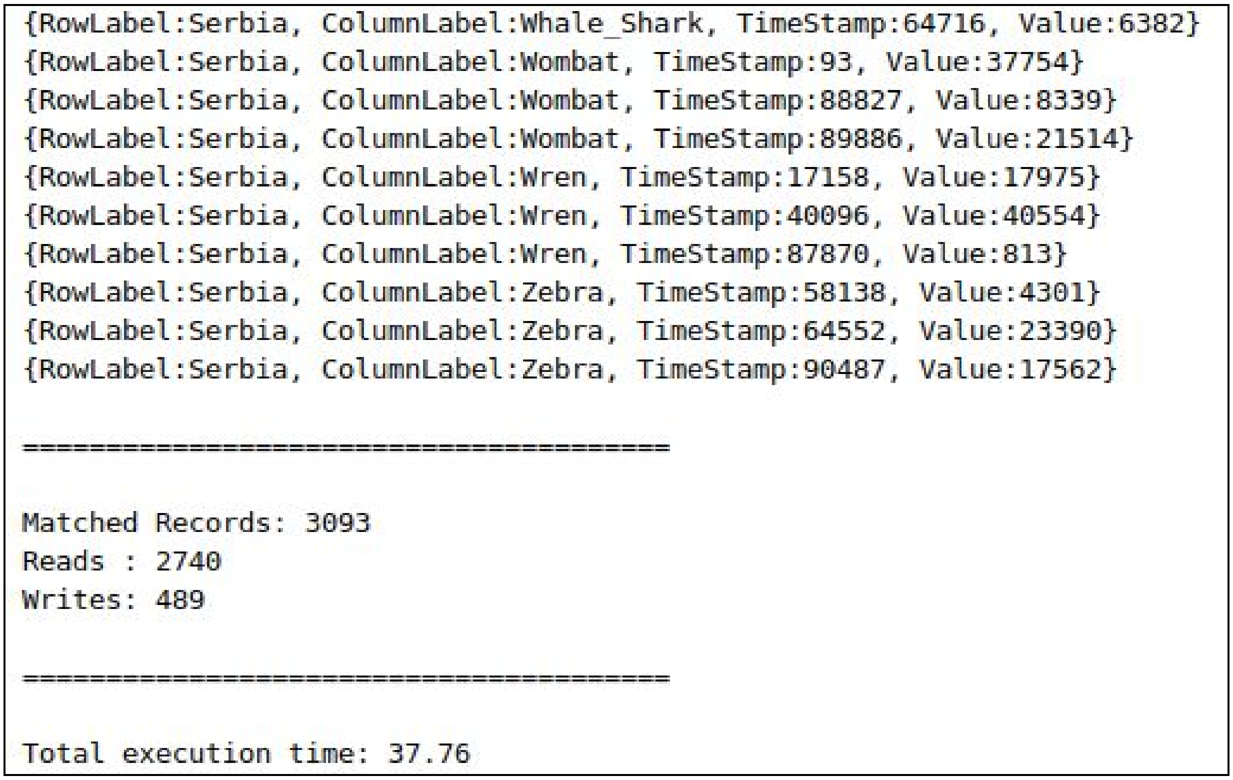
This query tests index type 1, order type 1, and row filter range functionalities.

query type1 1 1 [New_Jersey,Serbia] * * 200



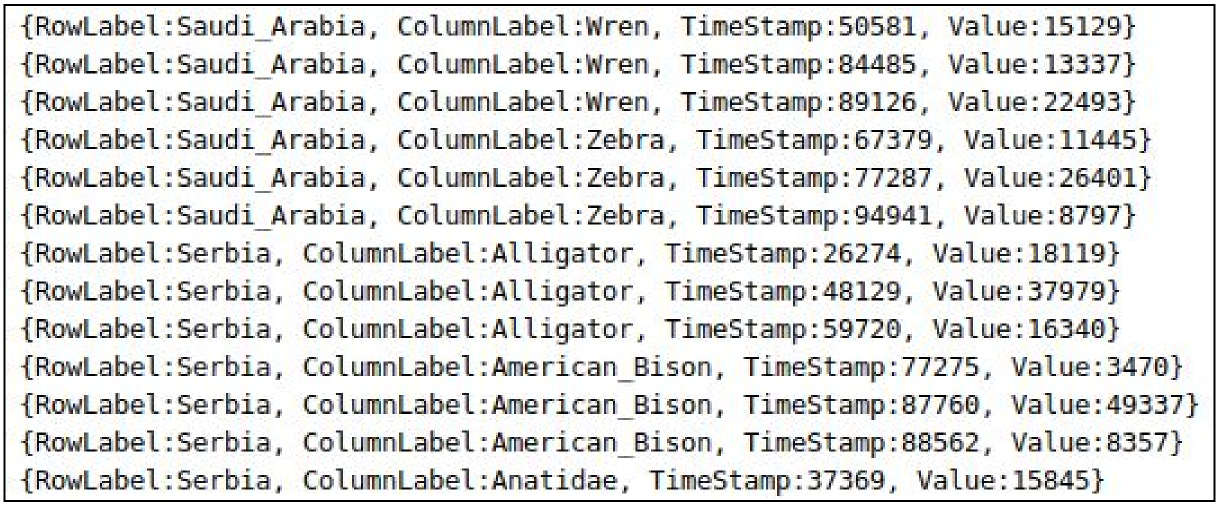
```
miniTable>  query type1 1 1 [New_Jersey,Serbia] * * 200
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Alligator, TimeStamp:41572, Value:280}
{RowLabel:New_Jersey, ColumnLabel:Alligator, TimeStamp:61558, Value:21301}
{RowLabel:New_Jersey, ColumnLabel:Alligator, TimeStamp:80822, Value:34797}
{RowLabel:New_Jersey, ColumnLabel:American_Bison, TimeStamp:25610, Value:28274}
{RowLabel:New_Jersey, ColumnLabel:American_Bison, TimeStamp:26246, Value:8149}
{RowLabel:New_Jersey, ColumnLabel:American_Bison, TimeStamp:47892, Value:36786}
{RowLabel:New_Jersey, ColumnLabel:Anatidae, TimeStamp:69345, Value:19039}
```

This output demonstrates that the query successfully implements a row filter and starts the range with “New Jersey” and ends with “Serbia” as demonstrated below.



```
{RowLabel:Serbia, ColumnLabel:Whale_Shark, TimeStamp:64716, Value:6382}
{RowLabel:Serbia, ColumnLabel:Wombat, TimeStamp:93, Value:37754}
{RowLabel:Serbia, ColumnLabel:Wombat, TimeStamp:88827, Value:8339}
{RowLabel:Serbia, ColumnLabel:Wombat, TimeStamp:89886, Value:21514}
{RowLabel:Serbia, ColumnLabel:Wren, TimeStamp:17158, Value:17975}
{RowLabel:Serbia, ColumnLabel:Wren, TimeStamp:40096, Value:40554}
{RowLabel:Serbia, ColumnLabel:Wren, TimeStamp:87870, Value:813}
{RowLabel:Serbia, ColumnLabel:Zebra, TimeStamp:58138, Value:4301}
{RowLabel:Serbia, ColumnLabel:Zebra, TimeStamp:64552, Value:23390}
{RowLabel:Serbia, ColumnLabel:Zebra, TimeStamp:90487, Value:17562}

========================================

Matched Records: 3093
Reads : 2740
Writes: 489

========================================

Total execution time: 37.76
```

Additionally, the Index Type 1 and Order Type 1 are successfully implemented with the sorting of Row Label, then Column Label, then Timestamp as demonstrated below with alphabetical ordering of row label “Saudi Arabia” to “Serbia”, column label “Wren” to “Zebra”, and increasing timestamp.



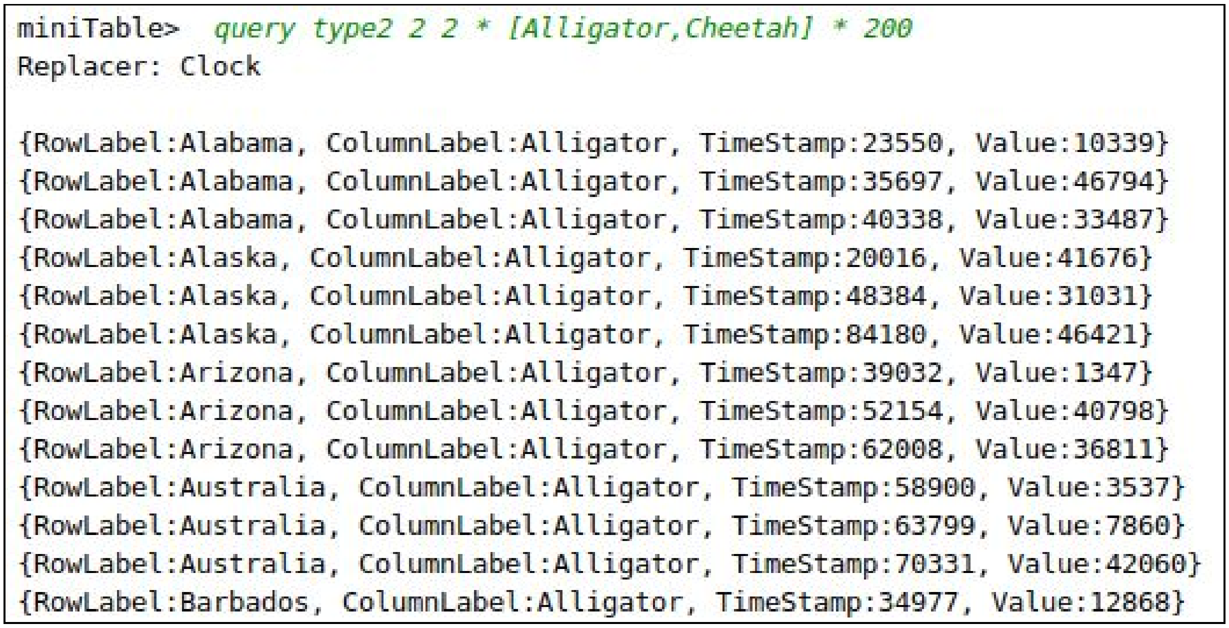
```
{RowLabel:Saudi_Arabia, ColumnLabel:Wren, TimeStamp:50581, Value:15129}
{RowLabel:Saudi_Arabia, ColumnLabel:Wren, TimeStamp:84485, Value:13337}
{RowLabel:Saudi_Arabia, ColumnLabel:Wren, TimeStamp:89126, Value:22493}
{RowLabel:Saudi_Arabia, ColumnLabel:Zebra, TimeStamp:67379, Value:11445}
{RowLabel:Saudi_Arabia, ColumnLabel:Zebra, TimeStamp:77287, Value:26401}
{RowLabel:Saudi_Arabia, ColumnLabel:Zebra, TimeStamp:94941, Value:8797}
{RowLabel:Serbia, ColumnLabel:Alligator, TimeStamp:26274, Value:18119}
{RowLabel:Serbia, ColumnLabel:Alligator, TimeStamp:48129, Value:37979}
{RowLabel:Serbia, ColumnLabel:Alligator, TimeStamp:59720, Value:16340}
{RowLabel:Serbia, ColumnLabel:American_Bison, TimeStamp:77275, Value:3470}
{RowLabel:Serbia, ColumnLabel:American_Bison, TimeStamp:87760, Value:49337}
{RowLabel:Serbia, ColumnLabel:American_Bison, TimeStamp:88562, Value:8357}
{RowLabel:Serbia, ColumnLabel:Anatidae, TimeStamp:37369, Value:15845}
```

Type Index 2

Order Type 2: Column Label, then Row Label, then Timestamp

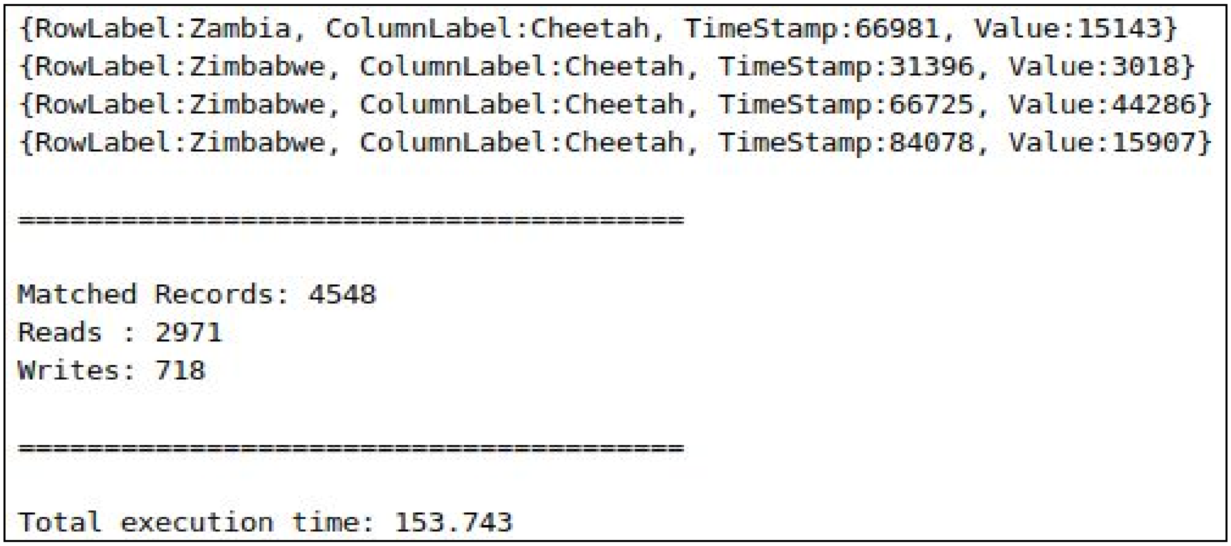
This query tests index type 2, order type 2, and column filter range functionalities.

query type2 2 2 * [Alligator,Cheetah] * 200



```
miniTable>  query type2 2 2 * [Alligator,Cheetah] * 200
Replacer: Clock

{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:23550, Value:10339}
{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:35697, Value:46794}
{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:40338, Value:33487}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:20016, Value:41676}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:48384, Value:31031}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:84180, Value:46421}
{RowLabel:Arizona, ColumnLabel:Alligator, TimeStamp:39032, Value:1347}
{RowLabel:Arizona, ColumnLabel:Alligator, TimeStamp:52154, Value:40798}
{RowLabel:Arizona, ColumnLabel:Alligator, TimeStamp:62008, Value:36811}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:58900, Value:3537}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:63799, Value:7860}
{RowLabel:Australia, ColumnLabel:Alligator, TimeStamp:70331, Value:42060}
{RowLabel:Barbados, ColumnLabel:Alligator, TimeStamp:34977, Value:12868}
```

This output demonstrates that the query successfully implements a column filter and starts the range with “Alligator” and ends with “Cheetah” as demonstrated below.



```
{RowLabel:Zambia, ColumnLabel:Cheetah, TimeStamp:66981, Value:15143}
{RowLabel:Zimbabwe, ColumnLabel:Cheetah, TimeStamp:31396, Value:3018}
{RowLabel:Zimbabwe, ColumnLabel:Cheetah, TimeStamp:66725, Value:44286}
{RowLabel:Zimbabwe, ColumnLabel:Cheetah, TimeStamp:84078, Value:15907}

=========================================

Matched Records: 4548
Reads : 2971
Writes: 718

=========================================

Total execution time: 153.743
```

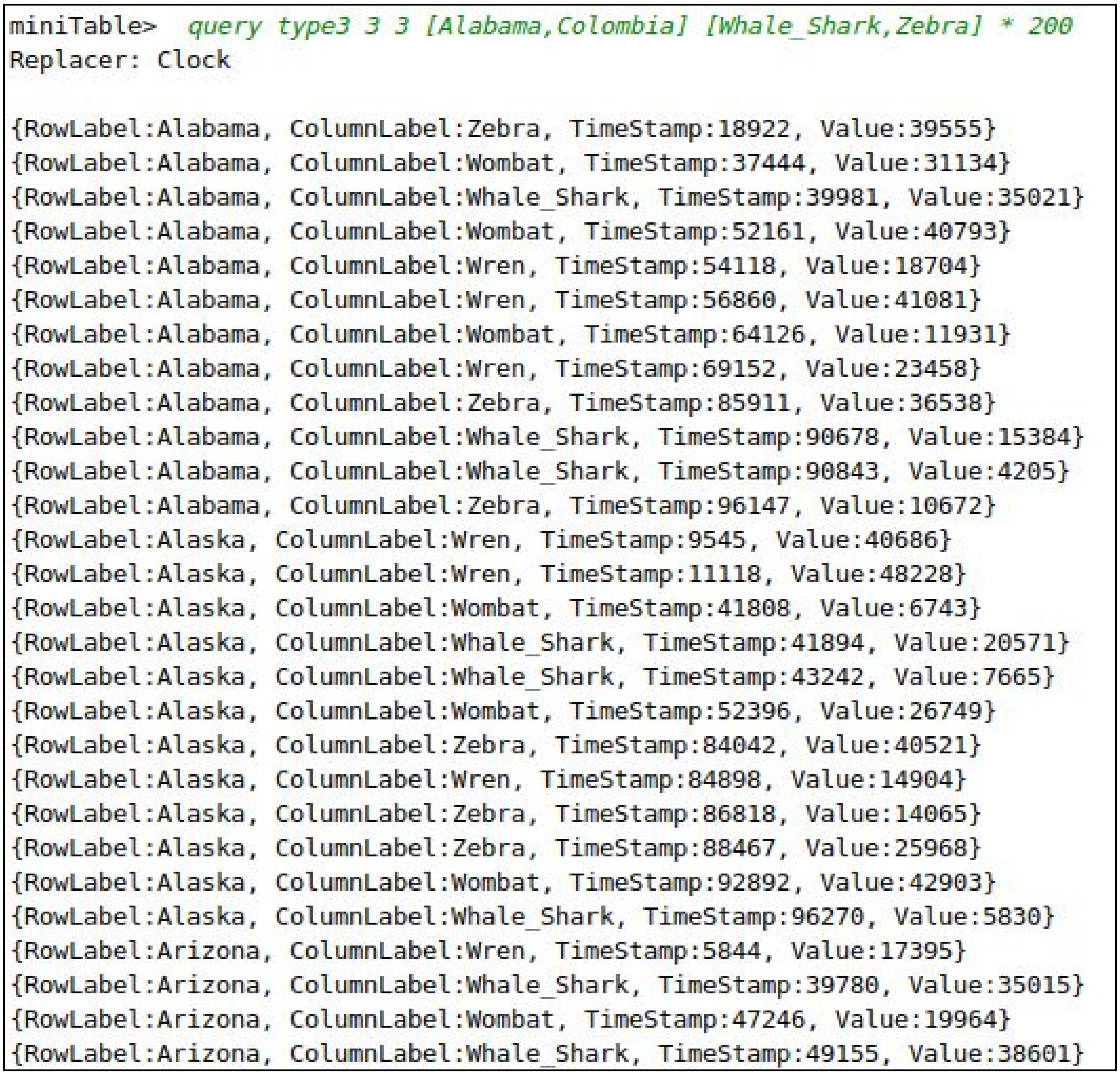
Additionally, the Indexing Type 2 and Order Type 2 are successfully implemented with the sorting of Column Label, then Row Label, then Timestamp as demonstrated with alphabetical ordering of column label “Alligator” to “Cheetah”, row label “Zambia” to “Zimbabwe”, and increasing timestamp.

Type Index 3

Order Type 3: Row Label, then Timestamp

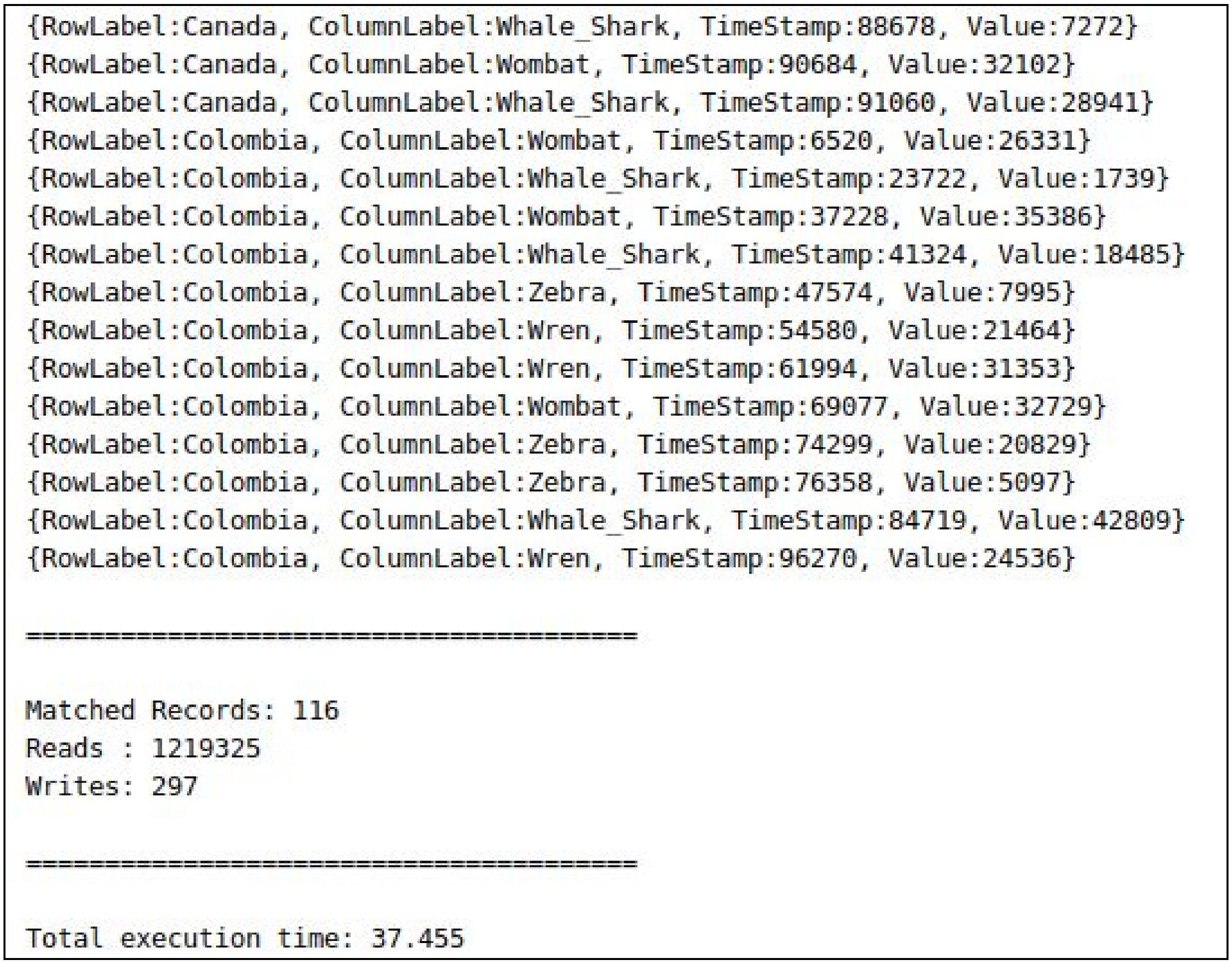
This query tests index type 3, order type 3, and multiple filters combined such as row filter range and column filter range functionalities.

```
query type3 3 3 [Alabama,Colombia] [Whale_Shark,Zebra] * 200
```



```
miniTable>  query type3 3 3 [Alabama,Colombia] [Whale_Shark,Zebra] * 200
Replacer: Clock

{RowLabel:Alabama, ColumnLabel:Zebra, TimeStamp:18922, Value:39555}
{RowLabel:Alabama, ColumnLabel:Wombat, TimeStamp:37444, Value:31134}
{RowLabel:Alabama, ColumnLabel:Whale_Shark, TimeStamp:39981, Value:35021}
{RowLabel:Alabama, ColumnLabel:Wombat, TimeStamp:52161, Value:40793}
{RowLabel:Alabama, ColumnLabel:Wren, TimeStamp:54118, Value:18704}
{RowLabel:Alabama, ColumnLabel:Wren, TimeStamp:56860, Value:41081}
{RowLabel:Alabama, ColumnLabel:Wombat, TimeStamp:64126, Value:11931}
{RowLabel:Alabama, ColumnLabel:Wren, TimeStamp:69152, Value:23458}
{RowLabel:Alabama, ColumnLabel:Zebra, TimeStamp:85911, Value:36538}
{RowLabel:Alabama, ColumnLabel:Whale_Shark, TimeStamp:90678, Value:15384}
{RowLabel:Alabama, ColumnLabel:Whale_Shark, TimeStamp:90843, Value:4205}
{RowLabel:Alabama, ColumnLabel:Zebra, TimeStamp:96147, Value:10672}
{RowLabel:Alaska, ColumnLabel:Wren, TimeStamp:9545, Value:40686}
{RowLabel:Alaska, ColumnLabel:Wren, TimeStamp:11118, Value:48228}
{RowLabel:Alaska, ColumnLabel:Wombat, TimeStamp:41808, Value:6743}
{RowLabel:Alaska, ColumnLabel:Whale_Shark, TimeStamp:41894, Value:20571}
{RowLabel:Alaska, ColumnLabel:Whale_Shark, TimeStamp:43242, Value:7665}
{RowLabel:Alaska, ColumnLabel:Wombat, TimeStamp:52396, Value:26749}
{RowLabel:Alaska, ColumnLabel:Zebra, TimeStamp:84042, Value:40521}
{RowLabel:Alaska, ColumnLabel:Wren, TimeStamp:84898, Value:14904}
{RowLabel:Alaska, ColumnLabel:Zebra, TimeStamp:86818, Value:14065}
{RowLabel:Alaska, ColumnLabel:Zebra, TimeStamp:88467, Value:25968}
{RowLabel:Alaska, ColumnLabel:Wombat, TimeStamp:92892, Value:42903}
{RowLabel:Alaska, ColumnLabel:Whale_Shark, TimeStamp:96270, Value:5830}
{RowLabel:Arizona, ColumnLabel:Wren, TimeStamp:5844, Value:17395}
{RowLabel:Arizona, ColumnLabel:Whale_Shark, TimeStamp:39780, Value:35015}
{RowLabel:Arizona, ColumnLabel:Wombat, TimeStamp:47246, Value:19964}
{RowLabel:Arizona, ColumnLabel:Whale_Shark, TimeStamp:49155, Value:38601}
```

This output demonstrates that the query is successfully able to handle multiple filters as combined input demonstrated by the “Alabama” to “Colombia” results and the “Whale Shark” to “Zebra” range results.



```
{RowLabel:Canada, ColumnLabel:Whale_Shark, TimeStamp:88678, Value:7272}
{RowLabel:Canada, ColumnLabel:Wombat, TimeStamp:90684, Value:32102}
{RowLabel:Canada, ColumnLabel:Whale_Shark, TimeStamp:91060, Value:28941}
{RowLabel:Colombia, ColumnLabel:Wombat, TimeStamp:6520, Value:26331}
{RowLabel:Colombia, ColumnLabel:Whale_Shark, TimeStamp:23722, Value:1739}
{RowLabel:Colombia, ColumnLabel:Wombat, TimeStamp:37228, Value:35386}
{RowLabel:Colombia, ColumnLabel:Whale_Shark, TimeStamp:41324, Value:18485}
{RowLabel:Colombia, ColumnLabel:Zebra, TimeStamp:47574, Value:7995}
{RowLabel:Colombia, ColumnLabel:Wren, TimeStamp:54580, Value:21464}
{RowLabel:Colombia, ColumnLabel:Wren, TimeStamp:61994, Value:31353}
{RowLabel:Colombia, ColumnLabel:Wombat, TimeStamp:69077, Value:32729}
{RowLabel:Colombia, ColumnLabel:Zebra, TimeStamp:74299, Value:20829}
{RowLabel:Colombia, ColumnLabel:Zebra, TimeStamp:76358, Value:5097}
{RowLabel:Colombia, ColumnLabel:Whale_Shark, TimeStamp:84719, Value:42809}
{RowLabel:Colombia, ColumnLabel:Wren, TimeStamp:96270, Value:24536}

=======================================

Matched Records: 116
Reads : 1219325
Writes: 297

=======================================

Total execution time: 37.455
```

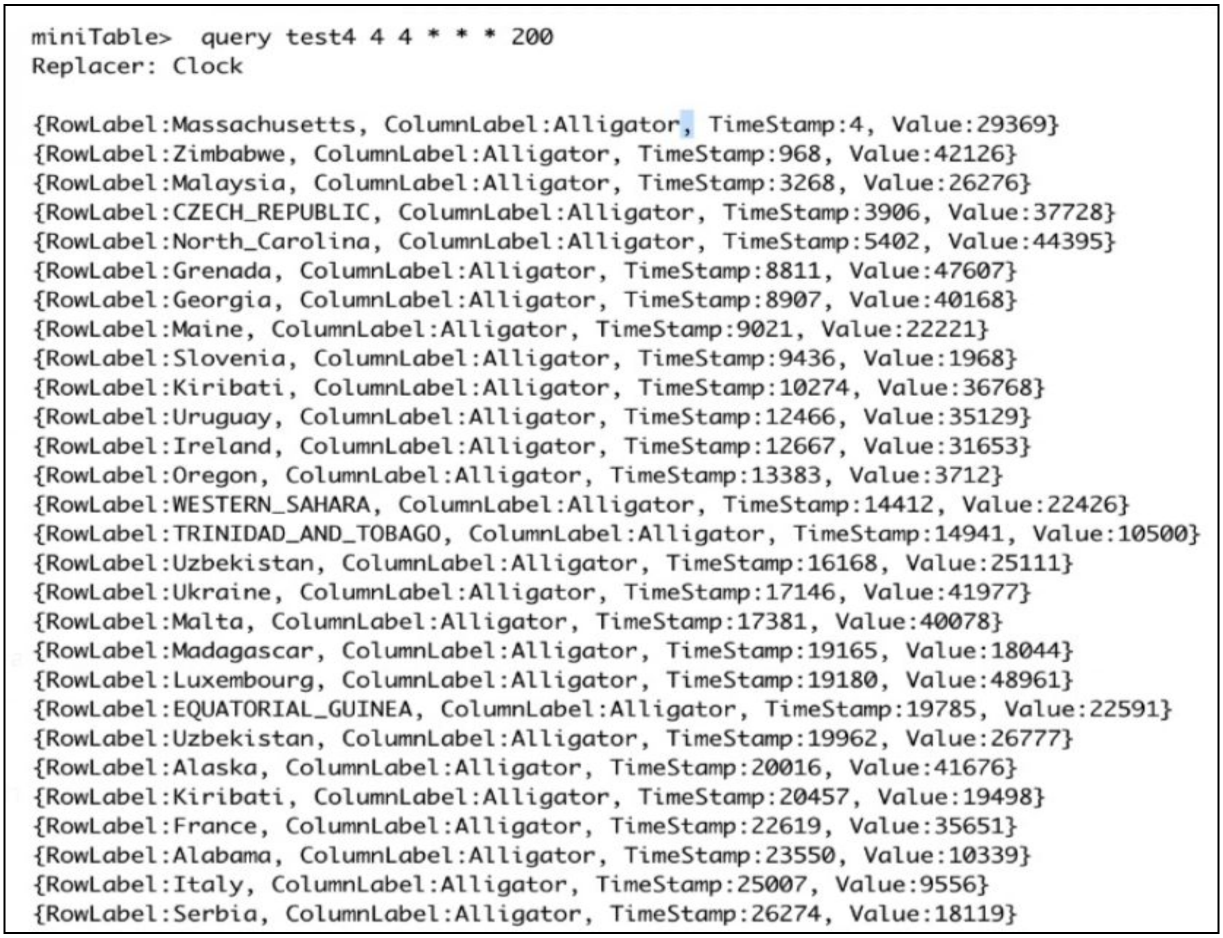
Additionally, the Indexing Type 3 and Order Type 3 are successfully implemented with the sorting of Row Label, then Timestamp as demonstrated with alphabetical ordering of row labels “Alabama” to “Colombia” and increasing timestamp. Meanwhile, the ordering of column labels is not alphabetical which shows that the order type 3 is working correctly.

Type Index 4

Order Type 4: Column Label, then Timestamp

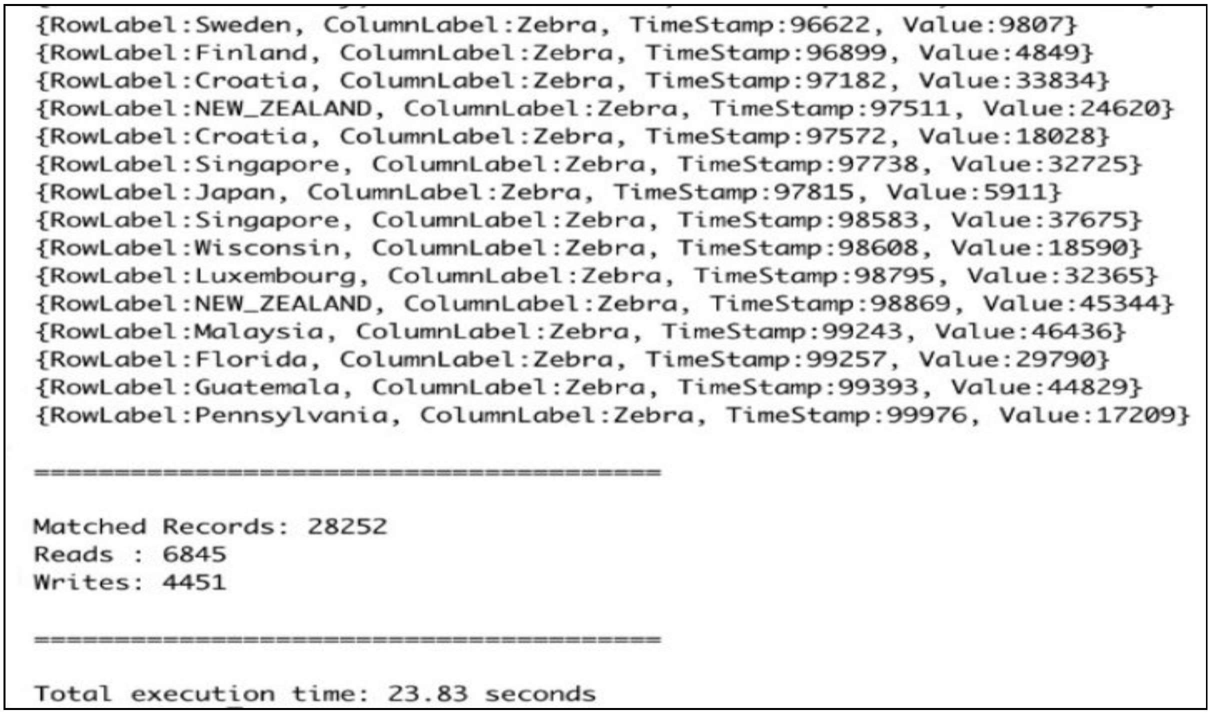
This query tests index type 4, order type 4, and unspecified functionalities for row filter and column filter, and value filter.

```
query test4 4 4 * * * 200
```



```
miniTable>  query test4 4 4 * * * 200
Replacer: Clock

{RowLabel:Massachusetts, ColumnLabel:Alligator, TimeStamp:4, Value:29369}
{RowLabel:Zimbabwe, ColumnLabel:Alligator, TimeStamp:968, Value:42126}
{RowLabel:Malaysia, ColumnLabel:Alligator, TimeStamp:3268, Value:26276}
{RowLabel:CZECH_REPUBLIC, ColumnLabel:Alligator, TimeStamp:3906, Value:37728}
{RowLabel:North_Carolina, ColumnLabel:Alligator, TimeStamp:5402, Value:44395}
{RowLabel:Grenada, ColumnLabel:Alligator, TimeStamp:8811, Value:47607}
{RowLabel:Georgia, ColumnLabel:Alligator, TimeStamp:8907, Value:40168}
{RowLabel:Maine, ColumnLabel:Alligator, TimeStamp:9021, Value:22221}
{RowLabel:Slovenia, ColumnLabel:Alligator, TimeStamp:9436, Value:1968}
{RowLabel:Kiribati, ColumnLabel:Alligator, TimeStamp:10274, Value:36768}
{RowLabel:Uruguay, ColumnLabel:Alligator, TimeStamp:12466, Value:35129}
{RowLabel:Ireland, ColumnLabel:Alligator, TimeStamp:12667, Value:31653}
{RowLabel:Oregon, ColumnLabel:Alligator, TimeStamp:13383, Value:3712}
{RowLabel:WESTERN_SAHARA, ColumnLabel:Alligator, TimeStamp:14412, Value:22426}
{RowLabel:TRINIDAD_AND_TOBAGO, ColumnLabel:Alligator, TimeStamp:14941, Value:10500}
{RowLabel:Uzbekistan, ColumnLabel:Alligator, TimeStamp:16168, Value:25111}
{RowLabel:Ukraine, ColumnLabel:Alligator, TimeStamp:17146, Value:41977}
{RowLabel:Malta, ColumnLabel:Alligator, TimeStamp:17381, Value:40078}
{RowLabel:Madagascar, ColumnLabel:Alligator, TimeStamp:19165, Value:18044}
{RowLabel:Luxembourg, ColumnLabel:Alligator, TimeStamp:19180, Value:48961}
{RowLabel:EQUATORIAL_GUINEA, ColumnLabel:Alligator, TimeStamp:19785, Value:22591}
{RowLabel:Uzbekistan, ColumnLabel:Alligator, TimeStamp:19962, Value:26777}
{RowLabel:Alaska, ColumnLabel:Alligator, TimeStamp:20016, Value:41676}
{RowLabel:Kiribati, ColumnLabel:Alligator, TimeStamp:20457, Value:19498}
{RowLabel:France, ColumnLabel:Alligator, TimeStamp:22619, Value:35651}
{RowLabel:Alabama, ColumnLabel:Alligator, TimeStamp:23550, Value:10339}
{RowLabel:Italy, ColumnLabel:Alligator, TimeStamp:25007, Value:9556}
{RowLabel:Serbia, ColumnLabel:Alligator, TimeStamp:26274, Value:18119}
```

This output demonstrates that the query is successfully able to handle multiple unspecified filters as combined input demonstrated by the “*” character and the result of 28252 records.



```
{RowLabel:Sweden, ColumnLabel:Zebra, TimeStamp:96622, Value:9807}
{RowLabel:Finland, ColumnLabel:Zebra, TimeStamp:96899, Value:4849}
{RowLabel:Croatia, ColumnLabel:Zebra, TimeStamp:97182, Value:33834}
{RowLabel:NEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:97511, Value:24620}
{RowLabel:Croatia, ColumnLabel:Zebra, TimeStamp:97572, Value:18028}
{RowLabel:Singapore, ColumnLabel:Zebra, TimeStamp:97738, Value:32725}
{RowLabel:Japan, ColumnLabel:Zebra, TimeStamp:97815, Value:5911}
{RowLabel:Singapore, ColumnLabel:Zebra, TimeStamp:98583, Value:37675}
{RowLabel:Wisconsin, ColumnLabel:Zebra, TimeStamp:98608, Value:18590}
{RowLabel:Luxembourg, ColumnLabel:Zebra, TimeStamp:98795, Value:32365}
{RowLabel:NEW_ZEALAND, ColumnLabel:Zebra, TimeStamp:98869, Value:45344}
{RowLabel:Malaysia, ColumnLabel:Zebra, TimeStamp:99243, Value:46436}
{RowLabel:Florida, ColumnLabel:Zebra, TimeStamp:99257, Value:29790}
{RowLabel:Guatemala, ColumnLabel:Zebra, TimeStamp:99393, Value:44829}
{RowLabel:Pennsylvania, ColumnLabel:Zebra, TimeStamp:99976, Value:17209}

=======================================

Matched Records: 28252
Reads : 6845
Writes: 4451

=======================================

Total execution time: 23.83 seconds
```

Additionally, the Indexing Type 4 and Order Type 4 are successfully implemented with the sorting of Column Label, then Timestamp as demonstrated with alphabetical ordering of column labels “Alligator” to “Zebra” and increasing timestamp. Meanwhile, the ordering of row labels is not alphabetical which shows that the order type 4 is working correctly.

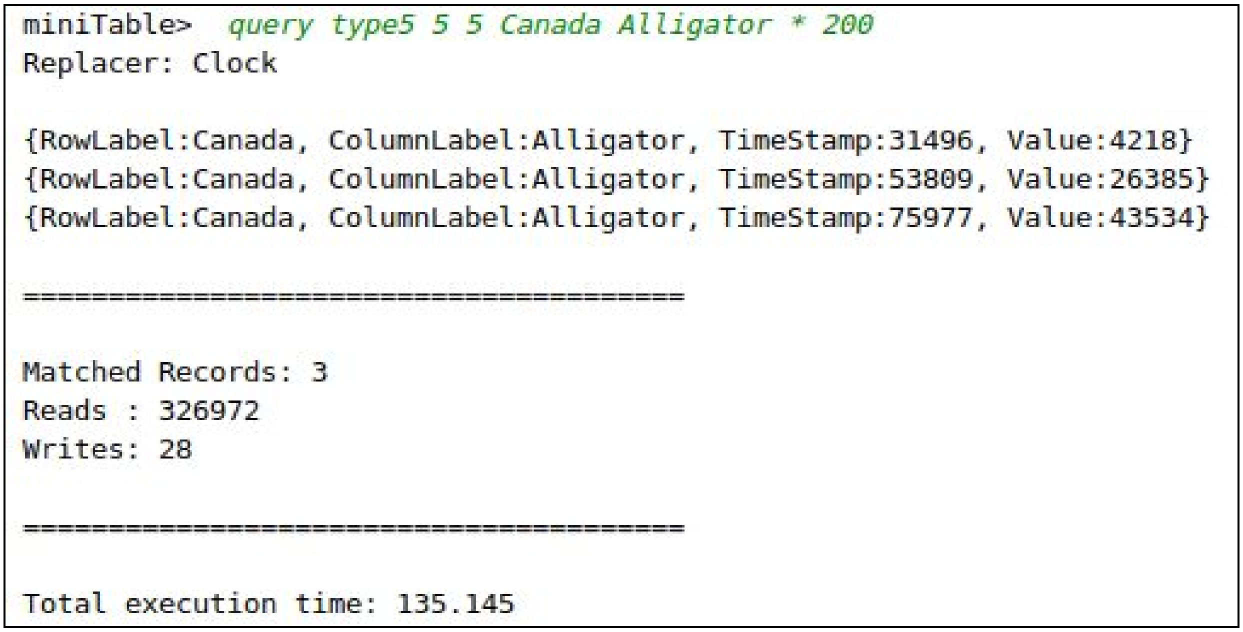
Type Index 5

Order Type 5: Timestamp

This query tests index type 5, order type 5, and single value functionalities for row filter and column filter.

query type5 5 5 Canada Alligator * 200

This output demonstrates that the query is successfully able to handle multiple single value filters as combined input demonstrated by the “Canada” results and the “Alligator” results.



```
miniTable>  query type5 5 5 Canada Alligator * 200
Replacer: Clock

{RowLabel:Canada, ColumnLabel:Alligator, TimeStamp:31496, Value:4218}
{RowLabel:Canada, ColumnLabel:Alligator, TimeStamp:53809, Value:26385}
{RowLabel:Canada, ColumnLabel:Alligator, TimeStamp:75977, Value:43534}

========================================

Matched Records: 3
Reads : 326972
Writes: 28

========================================

Total execution time: 135.145
```

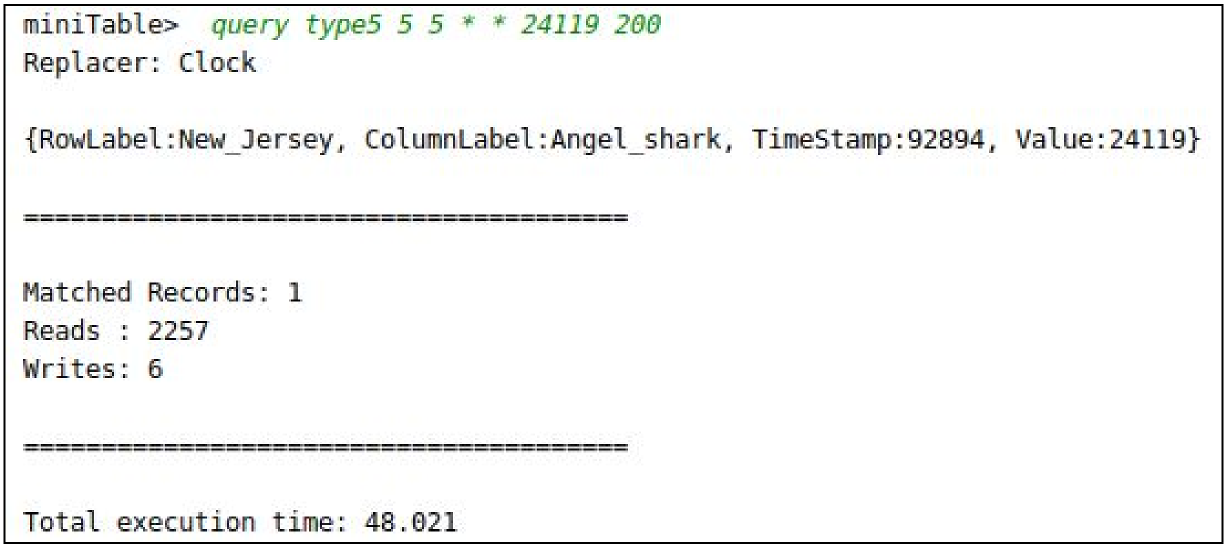
Additionally, the Indexing Type 5 and Order Type 5 are successfully implemented with the sorting of Timestamp as demonstrated with numerical ordering of timestamps.

Specific Value

This query tests index type 5, order type 5, and single value functionalities for value filter. It returns a record with one specific value.

```
query type5 5 5 * * 24119 200
```

This output demonstrates that the query is successfully able to return data with one specific value, “24119” shown in the results below.

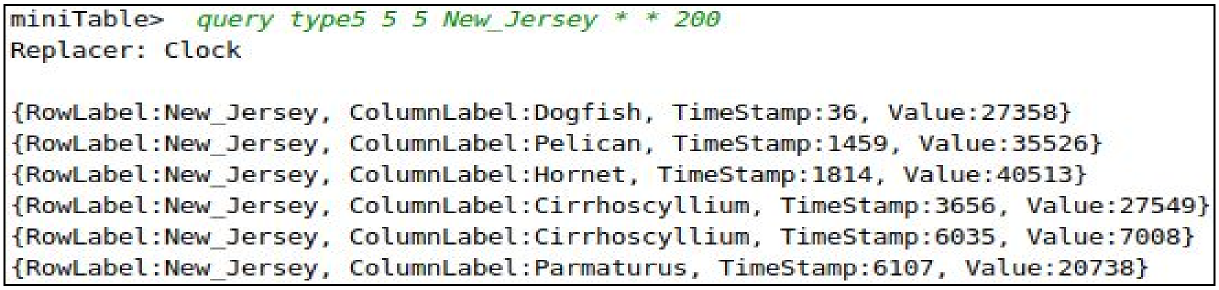


```
miniTable>  query type5 5 5 * * 24119 200
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Angel_shark, TimeStamp:92894, Value:24119}

==========================================

Matched Records: 1
Reads : 2257
Writes: 6

==========================================

Total execution time: 48.021
```

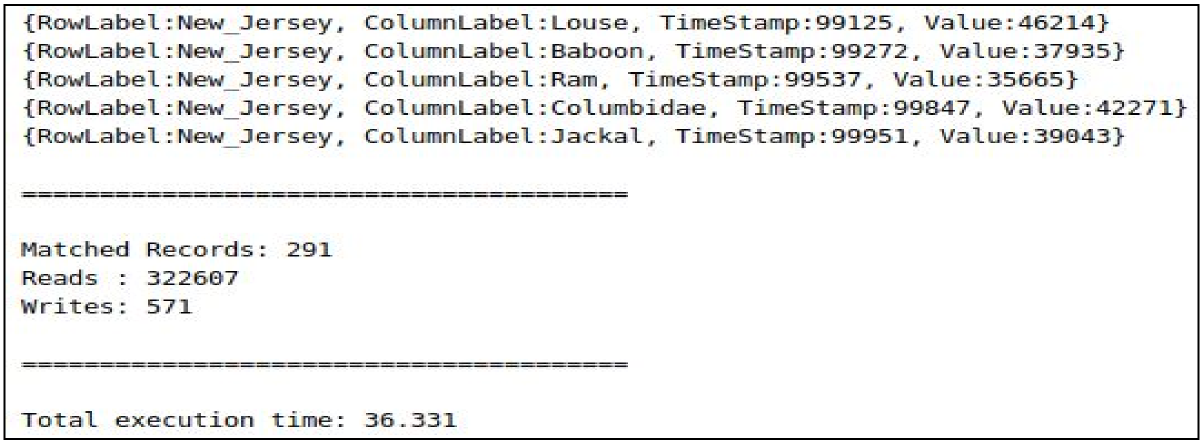
NUMBUF

This query tests different values of NUMBUF, the last parameter in the command, to analyze how the number of buffers affects performance including number of reads, number of writes, and total execution time. The different values of buffers tested include 200, 400, 1000, 1500, and 3000.

query type5 5 5 New_Jersey * * 200



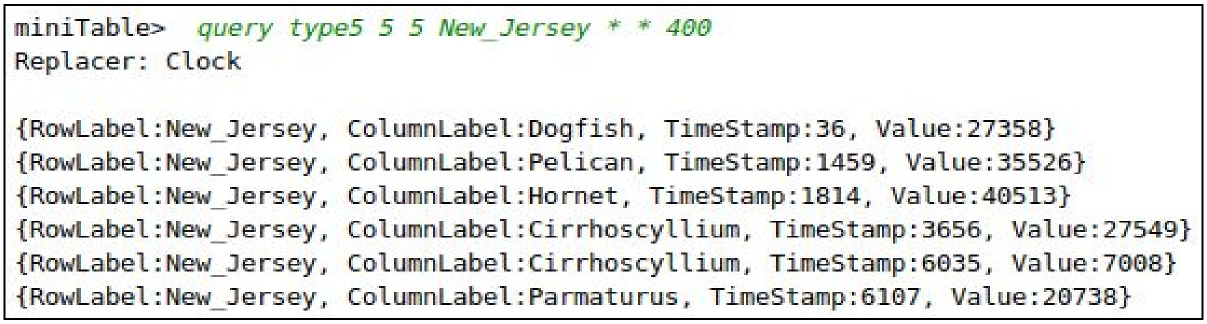
```
miniTable>  query type5 5 5 New_Jersey * * 200
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Dogfish, TimeStamp:36, Value:27358}
{RowLabel:New_Jersey, ColumnLabel:Pelican, TimeStamp:1459, Value:35526}
{RowLabel:New_Jersey, ColumnLabel:Hornet, TimeStamp:1814, Value:40513}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:3656, Value:27549}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:6035, Value:7008}
{RowLabel:New_Jersey, ColumnLabel:Parmaturus, TimeStamp:6107, Value:20738}
```



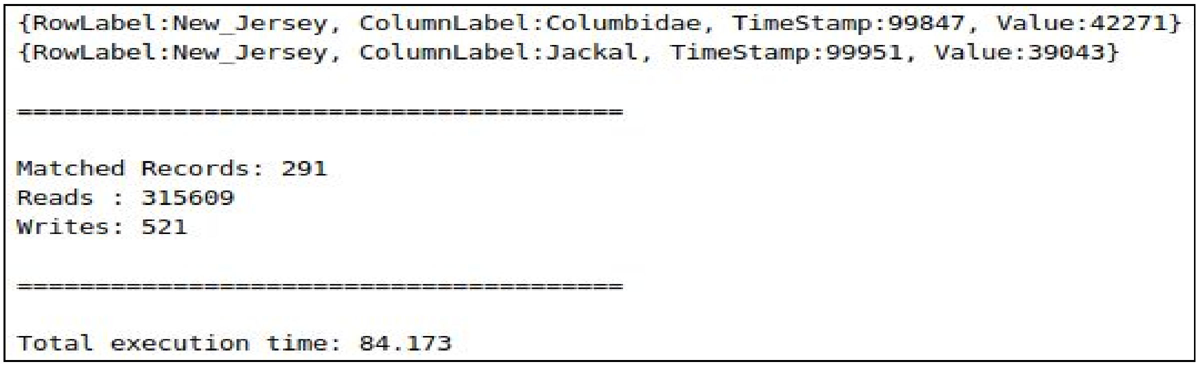
```
{RowLabel:New_Jersey, ColumnLabel:Louse, TimeStamp:99125, Value:46214}
{RowLabel:New_Jersey, ColumnLabel:Baboon, TimeStamp:99272, Value:37935}
{RowLabel:New_Jersey, ColumnLabel:Ram, TimeStamp:99537, Value:35665}
{RowLabel:New_Jersey, ColumnLabel:Columbidae, TimeStamp:99847, Value:42271}
{RowLabel:New_Jersey, ColumnLabel:Jackal, TimeStamp:99951, Value:39043}

==========================================

Matched Records: 291
Reads : 322607
Writes: 571

==========================================

Total execution time: 36.331
```

The output with 200 buffers demonstrates 291 matched records, 322607 reads, and 571 writes with an execution time of 36.331 seconds.

query type5 5 5 New_Jersey * * 400



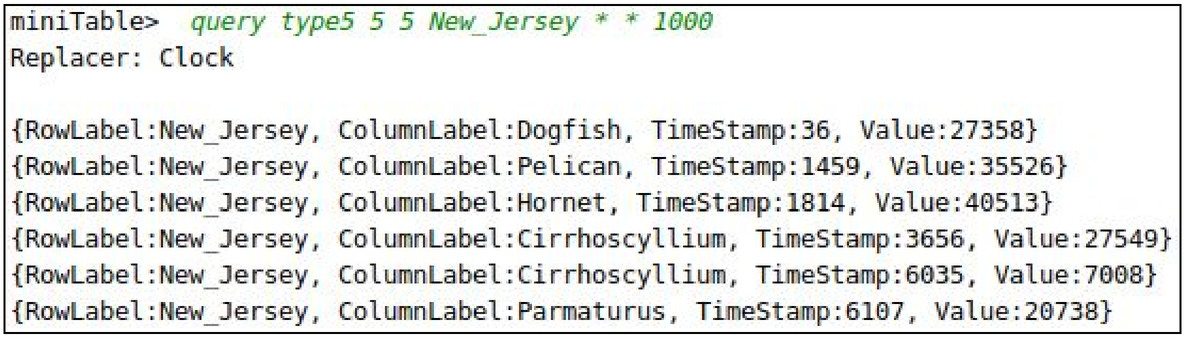
```
miniTable>  query type5 5 5 New_Jersey * * 400
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Dogfish, TimeStamp:36, Value:27358}
{RowLabel:New_Jersey, ColumnLabel:Pelican, TimeStamp:1459, Value:35526}
{RowLabel:New_Jersey, ColumnLabel:Hornet, TimeStamp:1814, Value:40513}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:3656, Value:27549}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:6035, Value:7008}
{RowLabel:New_Jersey, ColumnLabel:Parmaturus, TimeStamp:6107, Value:20738}
```



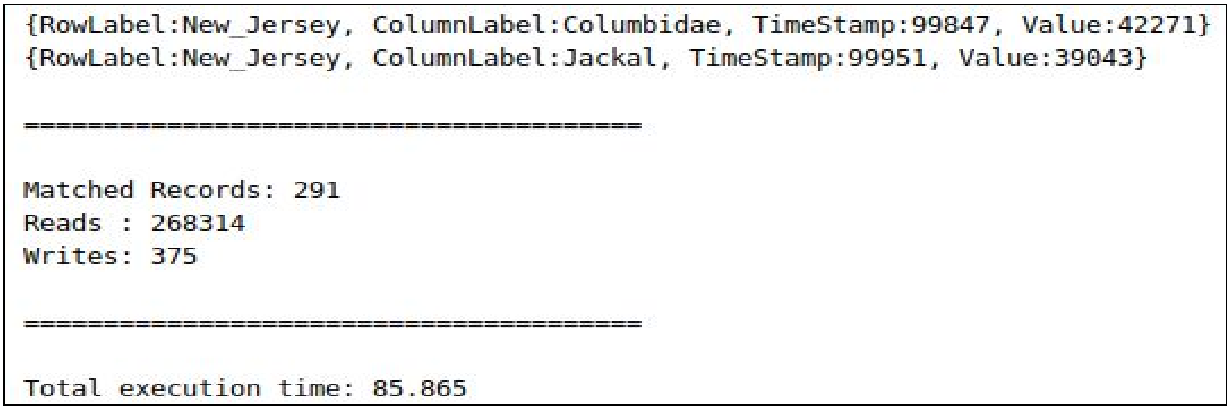
```
{RowLabel:New_Jersey, ColumnLabel:Columbidae, TimeStamp:99847, Value:42271}
{RowLabel:New_Jersey, ColumnLabel:Jackal, TimeStamp:99951, Value:39043}

=========================================

Matched Records: 291
Reads : 315609
Writes: 521

=========================================

Total execution time: 84.173
```

The output with 400 buffers demonstrates a decrease in number of reads with 315609 reads and a decrease in number of writes with 571 writes. The execution time however has increased more than double than previous with 84.173 seconds.

query type5 5 5 New_Jersey * * 1000



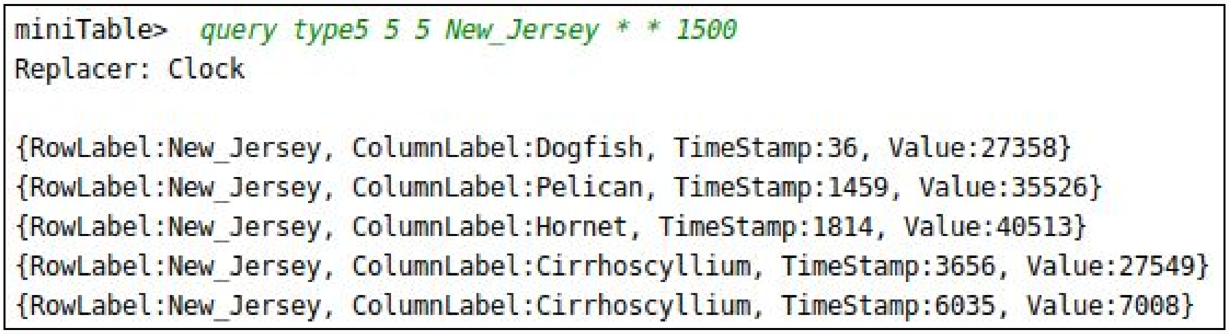
```
miniTable>  query type5 5 5 New_Jersey * * 1000
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Dogfish, TimeStamp:36, Value:27358}
{RowLabel:New_Jersey, ColumnLabel:Pelican, TimeStamp:1459, Value:35526}
{RowLabel:New_Jersey, ColumnLabel:Hornet, TimeStamp:1814, Value:40513}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:3656, Value:27549}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:6035, Value:7008}
{RowLabel:New_Jersey, ColumnLabel:Parmaturus, TimeStamp:6107, Value:20738}
```



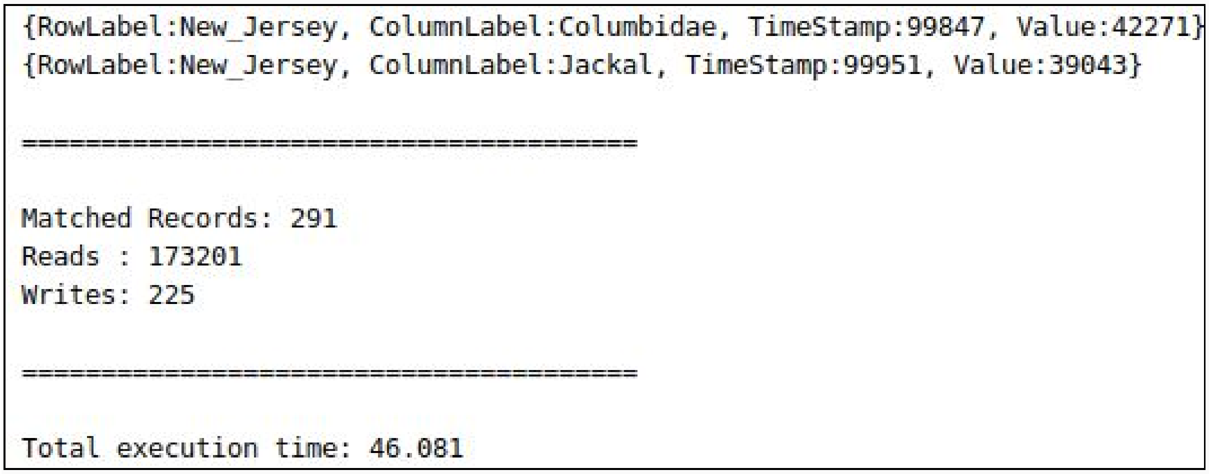
```
{RowLabel:New_Jersey, ColumnLabel:Columbidae, TimeStamp:99847, Value:42271}
{RowLabel:New_Jersey, ColumnLabel:Jackal, TimeStamp:99951, Value:39043}

=========================================

Matched Records: 291
Reads : 268314
Writes: 375

=========================================

Total execution time: 85.865
```

The output with 1000 buffers demonstrates a significant decrease in number of reads with 268314 reads and a decrease in number of writes with 375 writes. The execution time remains similar to the previous with 85.865 seconds. These results show good performance tradeoffs because the cost of disk accesses decreases while the execution time remains the same.

query type5 5 5 New_Jersey * * 1500



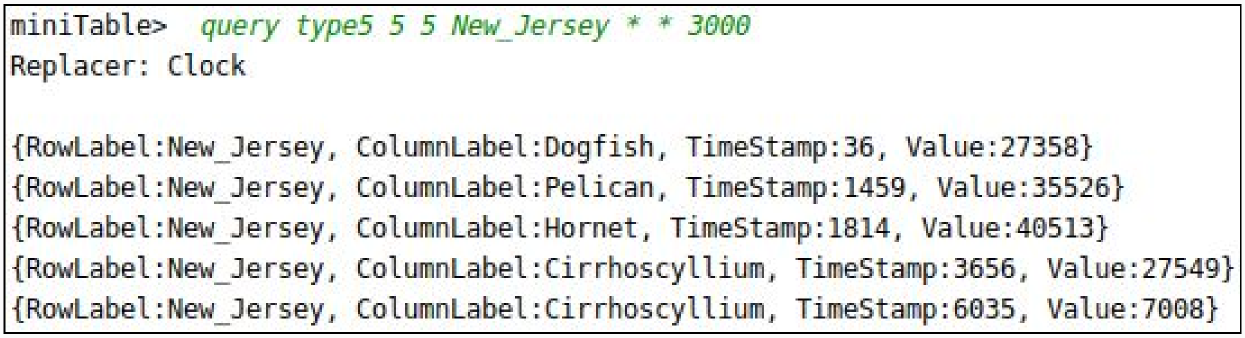
```
miniTable>  query type5 5 5 New_Jersey * * 1500
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Dogfish, TimeStamp:36, Value:27358}
{RowLabel:New_Jersey, ColumnLabel:Pelican, TimeStamp:1459, Value:35526}
{RowLabel:New_Jersey, ColumnLabel:Hornet, TimeStamp:1814, Value:40513}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:3656, Value:27549}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:6035, Value:7008}
```



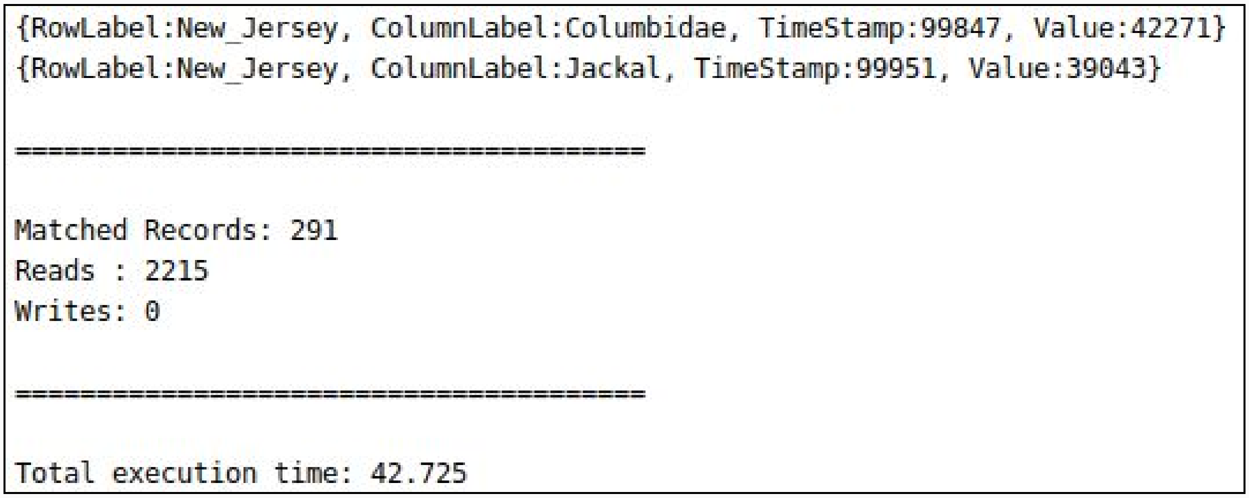
```
{RowLabel:New_Jersey, ColumnLabel:Columbidae, TimeStamp:99847, Value:42271}
{RowLabel:New_Jersey, ColumnLabel:Jackal, TimeStamp:99951, Value:39043}

=========================================

Matched Records: 291
Reads : 173201
Writes: 225

=========================================

Total execution time: 46.081
```

The output with 1500 buffers in comparison to 200 buffers demonstrates a 30% decrease in number of reads with 173201 reads and a 50% decrease in the number of writes with 225 writes. The execution time in comparison to 1000 buffers also decreases with 46.081 seconds. These results show strong performance tradeoffs because the cost of disk accesses decreases while the execution time also decreases. Overall, a lot of cost is saved.

query type5 5 5 New_Jersey * * 3000



```
miniTable>  query type5 5 5 New_Jersey * * 3000
Replacer: Clock

{RowLabel:New_Jersey, ColumnLabel:Dogfish, TimeStamp:36, Value:27358}
{RowLabel:New_Jersey, ColumnLabel:Pelican, TimeStamp:1459, Value:35526}
{RowLabel:New_Jersey, ColumnLabel:Hornet, TimeStamp:1814, Value:40513}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:3656, Value:27549}
{RowLabel:New_Jersey, ColumnLabel:Cirrhoscyllium, TimeStamp:6035, Value:7008}
```



```
{RowLabel:New_Jersey, ColumnLabel:Columbidae, TimeStamp:99847, Value:42271}
{RowLabel:New_Jersey, ColumnLabel:Jackal, TimeStamp:99951, Value:39043}

=======================================

Matched Records: 291
Reads : 2215
Writes: 0

=======================================

Total execution time: 42.725
```

The output with 3000 buffers in comparison to 200 buffers demonstrates a huge decrease in number of reads with 2215 reads and a very significant decrease in the number of writes with 0 writes. The execution time in comparison is also around the same with 42.725 seconds. These results show the best performance because the cost of disk accesses decreases while the execution time also decreases.

Overall, there is a more efficient impact on the performance with a significant enough increase in the number of buffers such as 3000 buffers vs 1000 buffers, while analyzing the tradeoffs between the amount of reads and writes and the execution time. However, adding too many buffers may also decrease the performance and add too much overhead. Therefore, finding the ideal threshold can be a trial and error process to provide the most benefit.

INTERFACE SPECIFICATIONS

Command Line Interface

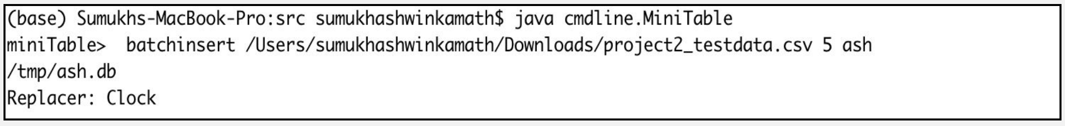
There are two commands that the user can execute to interact with the Big Table. They are listed below:

1. Batch Insert:

The command to execute the batch insert is:

```
batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
```

batchinsert	command used to execute batch insert
DATAFILENAME	file path of the csv which contains the maps to be inserted
TYPE	integer from 1 to 5 which indicates the clustering and indexing mechanism to be used for the Big table.
BIGTABLENAME	name of the Big Table that should be created



```
(base) Sumukhs-MacBook-Pro:src sumukhashwinkamath$ java cmdline.MiniTable
miniTable>  batchinsert /Users/sumukhashwinkamath/Downloads/project2_testdata.csv 5 ash
/tmp/ash.db
Replacer: Clock
```

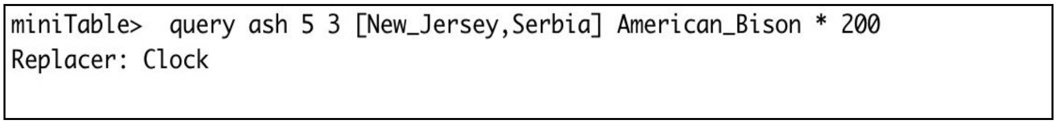
2. Query

The command to execute query is

```
query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER
NUMBUF
```

query	Command to execute query
BIGTABLENAME	name of the Big Table that should be created

TYPE	integer from 1 to 5 which indicates the clustering and indexing mechanism to be used for the Big table.
ORDERTYPE	Integer from 1 to 5 which indicates the type of ordering of the results from the query
ROW FILTER	Row filter
COLUMNFILTER	Column filter
VALUEFILTER	Value filter
NUMBUF	Number of buffers to be used for the query



```
miniTable>  query ash 5 3 [New_Jersey,Serbia] American_Bison * 200
Replacer: Clock
```

The filter can be of three types:

1. Matches all values(*)
2. A single value. Ex: New_Jersey
3. A range specified by two comma separated values in square brackets. Ex [New_Jersey,Serbia]

SYSTEM REQUIREMENTS

Operating system: Mac OSX 10.14 Mojave or higher, Ubuntu, Linux
Processor: 2.4 GHz 9th gen Intel Core-i5 or higher
Memory: 8GB or higher
System Type: 64 bit Operating System, x64 bit based processor
Java Version: 1.8.1
Language Level: 8

Executing the Program

1. Navigate to the source directory (`~<PROJECT_DIRECTORY>/MiniTable/src`).
2. Compile the source directory consisting of all related java class programs.

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

3. To execute the program run

```
<JDKPATH>/bin/java cmdline.MiniTable
```

This will provide a command line interface to run the commands for Batch Insertion and Querying.

Environments

Minibase Implementation is based on Linux based systems. As the project implementation is based on Minibase, it works well on Linux Environments.

Libraries

Minibase Distribution is used as the base Library and no external libraries are used. Only the native java packages are used.

RELATED WORK

This paper [3] introduces bigtable as a new type of distributed storage system. The system is primarily designed to store structured data and allowed scalability on a large scale. This system is used in many services provided by google namely, web indexing, Google Earth, and Google Finance. Each service varies in nature however the bigtable implementation is able to scale and provide a highly flexible approach to a multitude of problems. The bigtable is controlled via an API. The API provides functions for deleting tables and column families. Additionally, bigtable allows the use of regular expressions in order to return subsets of the column families.

Bigtable is designed by indexing a row value, column value and a timestamp. The data model used within the system groups rows into tablets this creates good locality when data is accessed. The columns on the other hands are grouped into column families. The number of distinct column keys is purposely maintained to be small. The final entity that is involved in the indexing is the timestamp, this allows for a history or a log of data. The timestamps allow for multiple entries to be stored, each with different values arranged in descending order, to always return the most recent first.

Bigtable is designed using three major components. These components interact with each other analogously to a master slave architecture. With a single master server and many tablet servers. At the top of the layer exists a Root tablet which consists of all the information needed in order to access each individual tablet. The master is in charge of maintaining all the tablets and tracking the assigned ones and unassigned ones. [3]

CONCLUSIONS

In this project, we have implemented a bigtable like (Key, Value) database on top of the minibase relational database. We created btree indexes on various keys and observed that the records could be fetched efficiently using these indexes. We performed batch inserts for varying buffer sizes and found that the number of disk reads and writes decreases, as we increase the number of buffers. We found an optimal number of buffers for our implementation. We also found that having a large number of buffer pages would not help much and reduces the performance. We also created various indexes during batch insert. The analysis shows various queries of different types like range search and equality search. By implementing this project, it gave us a very good insight on the internal working of the various components in a database management system. Playing with these internal components helped us gain more knowledge on the database management systems. In future we would like to consider optimizing the disk sorting algorithm to work with variable sized records.

BIBLIOGRAPHY

[1] http://pages.cs.wisc.edu/~dbbook/openAccess/Minibase/

REFERENCES

[1] *Database Management Systems*, R. Ramakrishnan and J. Gehrke. McGraw-Hill, Third Edition, 2003.
[2] Lecture Notes, *DBMS Implementation*, Dr. Selcuk Candan, Arizona State University, 2020.
[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data.7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), {USENIX} (2006), pp. 205-218.

APPENDIX

Team Member Roles

Team Member Name	Role/Tasks
Kanishk Bashyam	Built constructor for stream class, adding custom exceptions, Modified Mapscan.java file, Heap file modifications, Contributed to report, testing, integrating the code.
Sumukh Ashwin Kamath	Designing the map data structure, implementation of Map, bigDB, BigT, parts of query, indexing, batch insert, creation of temporary metadata hash structure, map versioning, debugging, testing and report
Baani Khurana	Project management, team management, and communication lead. Implemented counting of reads and writes. Heap file modifications, Conducted integrated testing of queries and batch inserts. Structured, organized, and contributed to the report.
Rakesh Ramesh	Designing the map data structure, implementation of Map, bigDB, BigT, parts of query, indexing, batch insert, creation of temporary metadata hash structure, map versioning, debugging, testing and report.
Sreenivasan Ramesh	Implementing cmdline, batch insert and querying, modifying other classes such as FileScan to use Maps, parts of Stream - Filtering records in Stream, indexing options, closeStream, BigT insertMap, integrating different branches, testing and report.
Ganesh Ashok Yallankar	Implementation of the Stream, MapSort, MapIterator, parts of Map, bigT, MapUtils, modifying dependent classes, validations on type and db, debugging, testing and report.