Project Phase II

Group 16

Members

Anjana Ouseph
Chandra Sai Chalicheemala
Lowkya Vuppu
Nagarjun Reddy Mora
Sai Anurag Vellanki

Abstract

Minibase is a database management system which is intended for educational purposes. This setup comes with a parser, optimizer, buffer pool manager, storage mechanisms such as heap files, secondary indexes based on B+ Trees, and a disk space management system. The goal of the minibase is not just to have a functional DBMS, but to have a DBMS where the individual components can be studied and implemented by students.

In phase II of this project, we made a bold attempt to use the modules of this RDBMS to build a Bigtable-like DBMS. We implemented the features of batch insert and query. In this report, we explain briefly the changes, and design decisions we made throughout this phase and output and statistics.

Problem Statement

In this phase of the project, we would be modifying the Minibase from a relational database management system to a Bigtable-like DBMS. We took the Phase II document that was provided as a reference and designed and implemented it accordingly. In this phase, we implemented batch insert and query functionalities.

Introduction

Minibase is a database management system which is intended for educational purposes. This setup comes with a parser, optimizer, buffer pool manager, storage mechanisms such as heap files, secondary indexes based on B+ Trees, and a disk space management system. The primary goal of this project is to build a Bigtable-like DBMS with features of batch insert and query.

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. Many projects at Google store data in Bigtable, including web indexing, Google Maps, Google Earth, and Google Finance.

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and timestamp; each value in the map is an uninterpreted array of bytes.

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

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp.

Assumptions

We assumed the following conditions during the implementation and the testing of the project.

1. According to the specifications of the Phase II specification document, we would be initializing the bigDB class with the type of index. So, we assumed that once it is initialized, only that type of inserts and queries would be done by the user.
2. The index type ranges from 1 to 5. So, we assume that we can expect the user to give the expected index types while using the commands in the interface.
3. The order type will be either 1, 2, 3, 4 or 6 as per the Phase II specification document.

Implementation and Proposed solution

Index Types : Below are the types and the reason why we chose these.

1.Value :
Faster queries on specific values: Instead of scanning the entire table for a given value, Bigtable uses an index on the value column.
Reduced Data Processing: By using this function, you can retrieve all rows in a query that contain a specific product name.
Improved Query Performance: Column indexes can improve query performance when a query is filtered or sorted by the value column. If you want to filter or sort a table by a particular value, adding an index to the value column can significantly improve query performance. An index based on the value column can help Bigtable locate specific rows without reading and processing unnecessary data.

2.RowLabel :
Faster row lookups : An index allows Bigtable to quickly find a specific row that matches a given row label. Individual row lookups can be significantly sped up, particularly for tables with many rows.
Reduced Data Processing: When locating a row using an index, Bigtable reads the entire row, as well as any related data, from the disk into memory.
Improved Query Performance: A query can be processed more efficiently by reducing disk seeks and improving data locality.

3.ColumnLabel :
Faster column scans: The index in Bigtable allows it to locate specific columns based on their label rather than having to search through the entire table. Particularly for tables with many columns, this can result in much faster column scanning.

Reduced data processing: Bigtable is able to avoid reading and processing unnecessary data when an index is used to locate specific columns.

Improved query performance: Indexes make it much easier for Bigtable to find the columns whose labels match a particular column label.

4.RowLabel and Column Label:

Faster lookups and scans: Using a compound index, Bigtable can quickly find the specific cells whose rows and columns match. For tables with many rows and columns, this can result in faster lookups and scans.

Reduced data processing: Bigtable can avoid reading and processing unnecessary data when a compound index is used to locate specific cells.

Improved query performance: Compound indexes enable Bigtable to quickly find cells that match both row and column labels, allowing you to construct queries that involve specific cells.

5.RowLabel and Value:

Faster queries: Instead of scanning through the entire table, Bigtable can locate matching rows quickly based on an index on both the row label and value columns. The result can be significantly faster query response times, especially for large tables.

Reduced data processing: Bigtable can avoid reading and processing unnecessary data when it uses an index to satisfy a query.

Improved data locality: Bigtable can read an entire row from disk into memory when a specific index is used to find it, along with any related data stored nearby on disk. To process a query, this can improve locality of data and reduce disk seeks.

Improved scalability: Despite the growth of the table, Bigtable can scale more effectively by indexing both row labels and value columns. The system becomes harder to scale without an index when the table size increases, making queries slower and more resource-intensive.

We created a new package called BigT and added the classes bigt, Map, Stream, BatchInsert, Query and Pair to this package.

bigT

We created this class to maintain the relevant heap files and index files for the table we are creating.

bigt(java.lang.String name, int type):

We use the constructor above to initialize the big table. The arguments here are passed by the BatchInsert class which is given by the user in the CLI. In the constructor, we initialize or fetch the relevant heap and index files. type here is an integer from 1 to 5. Depending on the type, you will be using different strategies for clustering and indexing the bigtable. The detailed description of index type and the reason to select those are mentioned below in the index types section.

void deleteBigt(): Deletes the Big Table. This method uses hf.deleteFileMap() to attempt to delete a file, and then utilityIndex.destroyFile() to destroy the file. It exits with a status code of 1 if any exception occurs during these operations.

int getMapCnt(): TotalMapCount is initialized to zero, and the method loops from 1 to 5 (inclusive). This method accessed the heap file object corresponding to the current index in the loop each iteration by calling the getRecCntMap() method. The totalMapCount variable is increased by the number of records in the heap file. A totalMapCount value is returned at the end of the loop, representing the total number of records (maps) in the five heap files.

int getRowCnt(): This method returns the number of distinct row labels in the bigtable.

int getColumnCnt(): This method returns the number of distinct column labels in the bigtable.

MID insertMap(byte[] mapPtr): It returns a MID instance representing the new record's identifier. The insertRecordMap method is used to obtain the MID object for the specified type. A record is inserted using the insertType variable with the specified type.

Stream openStream(int orderType, String rowFilter, String columnFilter, String valueFilter): By passing the rowFilter, columnFilter, and valueFilter parameters, the method creates a new instance of the Stream class. A Stream object can be easily created and configured using this method.

Map

Map():
This constructor creates a new byte array byte_data, initializes offset to 0, and sets len_map to max_maplength. If an input/output error occurs during initialization, the constructor may throw an IOException.

Map(byte[] amap, int offset):
The constructor sets byte_data equal to amap. As well, it takes in an offset value and sets this.offset to that value.

Map (Map fromMap):

The constructor initializes instance variables of the new object using those of the fromMap object. FromMap.getMapByteArray() returns byte_data equal to that byte array, offset is set to 0, len_map is set equal to the length of fromMap, and field_count and offset_field are set

accordingly. With the same data and field information, the new Map object is a copy of the fromMap object.

getRowLabel ():
Returns the row label of the Map object. A String representation of the byte data is generated using the Convert class. The resulting String is returned as the row label.

getColumnLabel():
Returns the column label of the Map object. A String representation is created using the Convert class. Column labels are returned as Strings

int getTimeStamp(): Returns the timestamp. This function returns the time stamp value of a Map object as an int. Bytes at the offset specified by offset_field array are converted to ints using the Convert class. Time stamp values are stored in the offset_field array, which contains the starting offset.

getValue(): Returns the value. This method returns the value of a Map object as a String. A String representation of the byte data at the offset specified by the offset_field array is created using the Convert class.

Map setRowLabel(String rowLabel): Sets the row label. Map objects are modified by the setRowLabel() method by setting the row labels to strings. The string is converted to bytes and stored in the appropriate offset position of the Map object's byte array.

Map setColumnLabel(String columnLabel):
This method sets the value of the column label in the byte array of the Map object. After converting columnLabel to bytes, bytes start at the offset specified in offset_field[1] in the byte array byte_data are written beginning at the offset of the columnLabel string. The updated Map object is returned at the end.

Map setTimeStamp(int timestamp):

In this method, the integer value of the time stamp field is converted to bytes and stored in the byte array at the offset positions. It then returns the updated Map object.

Map setValue(String value):
In the Map class, the setValue method sets the value field's value. This method takes a String parameter value and uses the Convert class to set the value in the byte_data array at offset_field[3]. It returns the Map object

byte[] getMapByteArray():

This method creates a new byte array with the same length as the original byte array and copies the contents of the original byte array beginning at the current offset in the new array. It then returns this new byte array.

void print():
The Map object's values are printed in a specific format. As a first step, it calls the getter methods for the row label, column label, timestamp, and value to retrieve their respective values. The value is then converted to a Long type and printed out as [rowLabel columnLabel timeStamp] -> value.

size():
Size() returns the number of bytes in the Map object. Using offset_field array, it subtracts starting offset from offset of the last field in the Map object.

mapCopy(Map fromMap):
MapCopy copies the contents of a Map object fromMap into the current Map object's byte_data array starting at the current offset. System.arraycopy() is used to copy the array.

mapInit(byte[] amap, int offset):

A byte array is set in the byte_data instance variable, and an integer offset is set in the offset instance variable. A pre-existing byte array can be used to initialize a Map object

mapSet(byte[] frommap, int offset):

This method sets the contents of the current Map object with the specified byte array from map starting at the specified offset and ending at len. Starting at index 0, it copies the contents of frommap into the byte_data array of the current object, sets the offset to 0 and sets the len_map to the specified length.

Stream

Stream(bigt bigtable, int orderType, String rowFilter, String columnFilter, String valueFilter):

The constructor for the Stream class sets up instance variables and takes several parameters. CondExpr objects are created to represent filter conditions for the big table, and index filter variables are initially set. In addition to indexType, the map_iterator object determines the type of scan to perform on the big table. After that, it initializes a SortMap object to sort the stream output and catches any exceptions.

void closestream(): Closes the stream object.

In the closestream() method, the sortMap object used for sorting the output of the stream is closed. Any exceptions thrown during the call to close() are caught by the method. In the event that an exception occurs while closing the stream, the method prints a message to the console.

Map getNext(MID mid):

GetNext() returns the next sorted Map object in the stream. If an exception is thrown during the process, the get_next() method is called on the sortMap object. Upon catching an exception, the method prints a message to the console and returns null.

Heap file changes

Heapfile(String name): There are two scenarios in which the constructor is called. In the first, the file is new and the header page must be initialized.
Failures in db->get_file_entry() are indicative of this case. It is only necessary to fetch the header page from the buffer pool in the second case when the file already exists

void deleteFileTuple(): Deletes the file from the database.

A first check is made to determine whether the file has already been deleted by checking the _file_deleted flag. True causes a FileAlreadyDeletedException to be thrown. In order to mark the file as deleted, the flag _file_deleted is set to true. A data page is then deallocated in the file. An initialization of the variable currentDirPageId is followed by initializing the variable with the ID of the first directory page.

Each directory page in the file is then looped through and all data pages linked to it are freed. The method calls freePage() to deallocate the data page for each tuple in the directory page. The method moves on to the next directory page once the current data page has been freed. All directory pages are freed until all are freed. The database's file directory is then cleaned up by calling the delete_file_entry() method.

boolean deleteMap(MID mid): Deletes map from file with given mid.

Data pages and directory pages where the record to be deleted is located are first searched. The method returns false if the record cannot be found. DataPageInfo entries on directories are updated if the record is found and deleted from the data page. The method deletes the corresponding DataPageInfo entry from the directory page if there are no more records on the data page. Depending on whether it is the first directory page or not, the method deletes or unpins the directory page if it becomes empty.

Map getMapRecord(MID mid): MID records are retrieved from a heap file and returned as Map objects with this method. MID page number is obtained by comparing it to the current page in memory, and then it is compared to the MID page number. After checking that the slot number is valid, it retrieves the length of the record from the slot and copies it into a new byte array if the length matches the slot number. The byte array is then converted into a Map object and returned.

MID insertRecordMap(byte[] recPtr) Inserts map into file, returns its Mid. In order to accommodate the new record, it first checks the Heap File for a data page with sufficient space. New records are inserted into such pages if they exist. The DataPageInfo record of the directory page of the Heap File is inserted if a suitable page cannot be found.

Scan openScan(): Initiates a sequential scan. In the openScan method, an instance of the class with the openScan method represents a relation that is scanned through using the Scan object.

boolean updateRecordMap(MID mid, Map newmap): Updates the specified map in the heapfile. A record update is performed by calling the _findDataPageMap method. An update that fails due to the inability to locate the record returns false. In the event the record is found, the old map will be retrieved. As soon as the new map's length does not match the old map's length, an exception is thrown and false is returned. The method otherwise writes the updated map back to the page and updates the old map with the new map. Data and directory pages are unpinned by this method.

Map utils

static int CompareMapWithMap(Map m1, Map m2, int map fld no):

A comparison between a map m1 and another map m2 in the respective field is performed by this function. It returns: 0 if the two maps are equal, 1 if the first map is greater, and -1 if the first map is smaller.

static boolean Equal(Map m1, Map m2): This function compares two maps in all fields. It returns 0 if the two maps m1 and m2 are not equal and returns 1 if they're equal.

bigDB

bigDB.java is created by modifying the existing DB.java file in the diskmgr package. Data organization is handled by this class by creating and maintaining all relevant files and btree based index files.

bigDB(int type): It's a default constructor in BigDB which takes type as a parameter and initializes the PCounter.

MID Changes

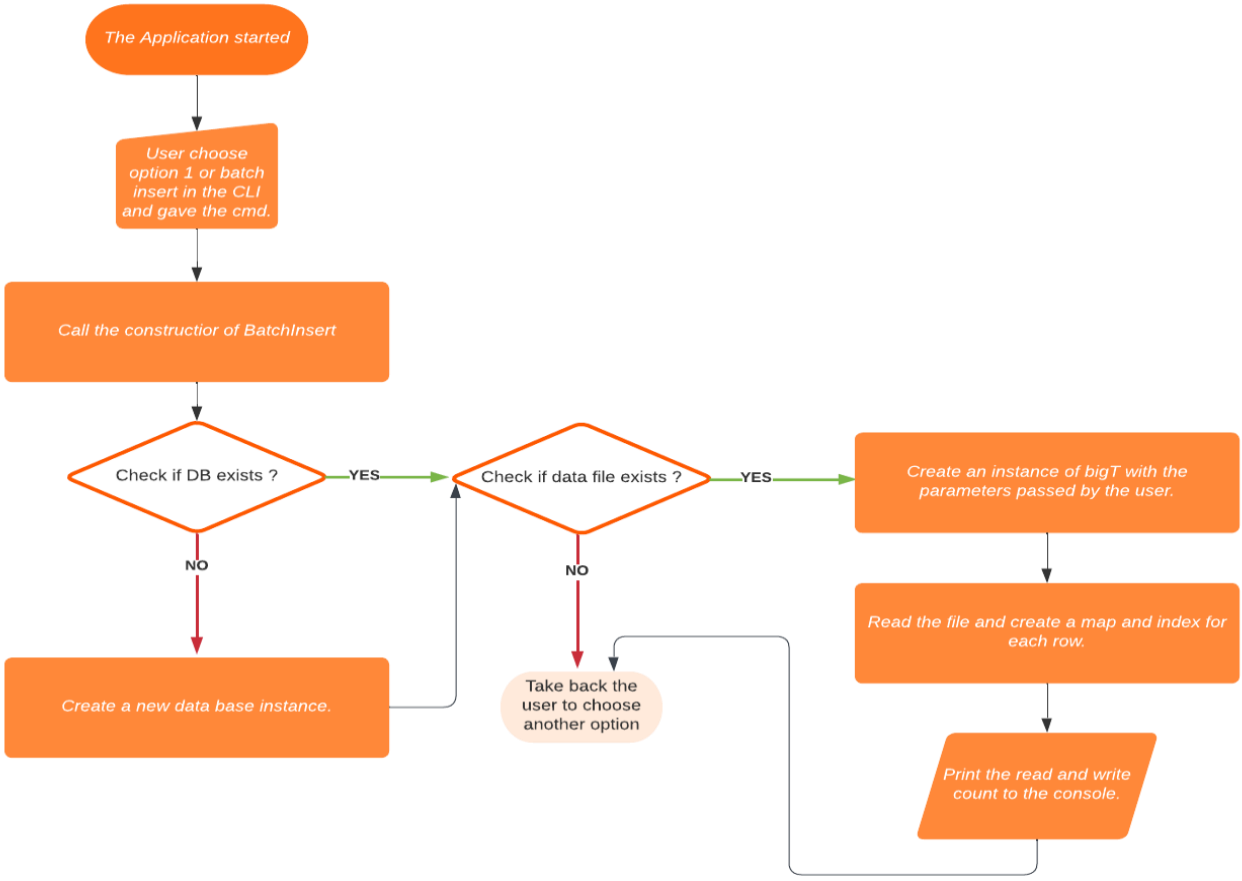
global.MID

MID is the map id which consists of a pageid and slotid to identify a specific map entry. In this phase of the project, MID replaces RID from the minibase so the files that have references of RID are modified and replaced with MID. RID references are removed in all the packages(btree,bufmgr, catalog, heap, index and iterator packages). In the btree package RID was used to get the position in the current leaf as RID, as key /RID was used in the leaf as a pointer. This reference of RID was changed to MID and now to get the current and next leaf i.e., map MID is used. Insertrecord() method returns MID at where the entry was inserted, inorder to record its pointer to pass it in the getnext() to get the next map. Similarly in all the packages where was RID used to be the reference to the getnext() and leafData() methods all the those where changed to MID. In the iterator packages this MID is used as param to getNextTuple() method which was RID earlier. Therefore as required all the RID references in the inserttion, getnext tuple or map method were changed to MID.

PCounter

We added this class to maintain the page reads and writes that the database does. This class has two static instance variables - rCounter and wCounter, which keep track of read and write count respectively. We call the static method to initialize()to set these counters to 0 and call the static methods readIncrement() and writeIncrement() to increment the
rCounter and wCounter respectively. These methods are called from the read_page() and write_page() of bigDB class.

BatchInsert



Flow chart of batch insert

public BatchInsert(String datafile, int type, String bigTableName, int numbuf)

Constructor of this class, initializes the database if it is not already initialized using the SystemDefs class with the signature as follows.

public SystemDefs(String dbname, int num_pgs, int bufpoolsize, String replacement_policy)

We pass the path for creating the database (to /tmp directory), number of pages, buffer pool size (from the input of batch insert) and the replacement policy (Clock).

Once the database is created or confirmed that it exists, we then proceed to check if the data file exists. If it doesn't we raise an exception and display the message to the user and ask the user to choose the options again. If it does, we proceed to create the bigT instance and read the CSV file and insert the data using Map and create the index for this. Once the process completes, we print the statistics to the user which includes the read count, write count and the number of records inserted.

Query

Implemented a program query that accesses the Bigtable Database and prints the maps in the requested order.
The command line invocation is of the below format

query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF

When the application starts, it asks the user to batchinsert as described below. After the bigtable is successfully created from the datafile, the user is taken back to the menu options. Here the user inputs option 2: query.

The user is required to input the below:

BIGTABLENAME: Name of the bigtable the user wants to query from.

TYPE: the type which defines which heap file and index file should be used to insert or fetch the data. The batch insert operation inserts the data into the corresponding heap and index files for a specific type and the query operation will fetch the data from the same heap and index files.

ORDERTYPE: It defines the sorting order of the output. The output will be sorted as per the below sorting orders.
If orderType = 1, then results are first ordered in row label, then column label, then timestamp · if orderType =2, then results are first ordered in column label, then row label, then timestamp · If orderType =3, then results are first ordered in row label, then time stamp.
If orderType = 4, then results are first ordered in column label, then time stamp.
If orderType = 6, then results are ordered in time stamp.
Since it's not mentioned in the project phase 2 document what orderType = 5 is supposed to do, we have assumed it sorts the results first in row label and then values.

ROWFILTER, COLUMNFILTER, VALUEFILTER: These are the filters which we need to give for rows, columns and values.
Each filter can be

"*": meaning unspecified (need to be returned),
a single value, or
a range specified by two column separated values in square brackets (e.g. "[56, 78]")
If any of the filter are null strings, then that filter is not considered.
The program will return the output based on the filter values which we provide as well as the sorting order.

NUMBUF: This is the value of the buffer pages which the user needs to pass. Minibase will use at most NUMBUF buffer pages to run the query.

public Query(String bigtName, int type, int orderType, String rowFilter, String columnFilter, String valueFilter, int numBuf)

Constructor of the class Query, we create a bigT instance with the bigtName and type. Then we call the openStream method of Stream class to retrieve the data from the database based on the rowfilter, column filter and value filter values and the sorting order.

Once the process completes, we print the statistics to the user which includes the number of records, read count, write count with the help of the PCounter class.

Performance Analysis

BatchInsert statistics

The table below shows the read and write counts for 250, 500, 750 and 1000 buffers for the 5 index types we choose. We noticed the trend that as the number of buffers increases, the read and write count goes down.

	BC : 250	BC : 500	BC : 750	BC : 1000
Type 1	RC: 125 WC: 810	RC: 125 WC: 560	RC: 125 WC: 310	RC: 125 WC : 60
Type 2	RC: 498 WC: 1589	RC: 186 WC : 1027	RC: 157 WC : 748	RC: 142 WC : 483
Type 3	RC: 468 WC : 1535	RC: 200 WC : 1017	RC: 151 WC : 718	RC: 144 WC : 461
Type 4	RC: 4656 WC : 5768	RC: 807 WC : 1674	RC: 191 WC : 808	RC: 139 WC : 506
Type 5	RC: 719 WC : 1936	RC: 189 WC : 1156	RC: 153 WC : 870	RC: 150 WC : 617

Query statistics

The table below shows the read and write counts for 250 and 500 buffer size for the 5 index types we chose and also for the 5 Order Types (1, 2, 3 ,4 and 6). We haven't taken readings for Order Type 5 as it is not defined in the phase 2 description. We noticed the trend that as the number of buffers increases, the read and write count goes down. Also, the general trend is from type 2 index onwards, the read and write counts are staying constant for a particular Order Type.

	Order Type 1	
	BC : 250	BC : 500
Type 1	RC: 122244 WC: 2009	RC: 1989 WC: 1553
Type 2	RC: 122121 WC: 2009	RC: 1990 WC: 1554
Type 3	RC: 122121 WC : 2009	RC: 1990 WC: 1554
Type 4	RC: 122121 WC : 2004	RC: 1990 WC: 1554
Type 5	RC: 122121 WC : 2008	RC: 1990 WC: 1554

	Order Type 2	
	BC : 250	BC : 500
Type 1	RC: 122365 WC: 1888	RC: 2112 WC: 931
Type 2	RC: 122242 WC: 1887	RC: 2113 WC: 931
Type 3	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 4	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 5	RC: 122242 WC : 1887	RC: 2113 WC: 931

	Order Type 3	
	BC : 250	BC : 500

Type 1	RC: 122365 WC: 1888	RC: 2112 WC: 931
Type 2	RC: 122242 WC: 1887	RC: 2113 WC: 931
Type 3	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 4	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 5	RC: 122242 WC : 1887	RC: 2113 WC: 931

	Order Type 4	
	BC : 250	BC : 500
Type 1	RC: 122365 WC: 1888	RC: 2112 WC: 931
Type 2	RC: 122242 WC: 1887	RC: 2113 WC: 931
Type 3	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 4	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 5	RC: 122242 WC : 1887	RC: 2113 WC: 931

	Order Type 6	
	BC : 250	BC : 500
Type 1	RC: 122365 WC: 1888	RC: 2112 WC: 931
Type 2	RC: 122242 WC: 1887	RC: 2113 WC: 931
Type 3	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 4	RC: 122242 WC : 1887	RC: 2113 WC: 931
Type 5	RC: 122242 WC : 1887	RC: 2113 WC: 931

Interface Specifications

```
----------------------- BigTable Tests -------------------------
Press 1 for Batch Insert
Press 2 for Query
Press 3 for MapInsert
Press 4 for RowJoin
Press 5 for RowSort
Press 6 for getCounts
Press 7 for other options
Press 8 to quit
----------------------- BigTable Tests -------------------------
```

User can interact with this application only using the CLI (Command Line Interface) which gets started by following the steps listed in the next section. For batch insert, choose the option 1 from the menu displayed. You will be presented with the expected format for the command. Similarly, for query choose option 2. Choose option 8 to quit the menu.

System requirements/installation and execution instructions

A system running on Java 17.

Steps to run the application:

1. Download the src zip
2. Change the src/makefile to set ASSIGN to your corresponding src directory.
3. Make sure that the $(JAVA_HOME) points to the jdk path of your machine.
4. Then run make main inside src folder to start the application. NOTE : test file is present in the code src folder named as test_data1.csv.

Conclusions

For phase II of this project, we have successfully converted the relational database to a big table as per the requirements specified. We came up with five clusterings and indexing schemes for this implementation. We modified the existing logic to reflect the new type of database. We implemented a batch insert and query feature that illustrates our effort. The project was thoroughly tested by running queries against all indexing schemes with the provided test data. We calculated the read and write counts for each operation and printed it on the console. We were able to observe the variations of read and write for different

indexing schemes. Overall, this project helped us gain a deeper understanding of the indexing schemes and gain practical knowledge about Google's big table.

Bibliography

https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
https://us.edstem.org/api/resources/24610/download/5%20-%20phase%202%20project%20description.pdf

Appendix

This was a collaborative effort between the five of us, there was a lot of overlap during the development and testing phases. Collectively, as a group, we believed we achieved what we set out to do for the Phase II of this project. The table below describes the primary contribution of each member.

Feature/Implementations/Tasks	**Primary developer**	**Secondary developer**
Pushing the base code to GitHub	Nagarjun Reddy Mora	
BigT package with classes	Nagarjun Reddy Mora	
bigt and Map classes.	Nagarjun Reddy Mora	
BigT.Stream	Chandra Sai Chalicheemala	
Map Utils, DiskMgr	Sai Anurag Vellanki	
Big DB	Nagarjun Reddy Mora	
PCounter	Lowkya Vuppu	
Batch Insert	Lowkya Vuppu	
Query	Anjana Ouseph	
MID Changes	Sai Anurag Vellanki	
Report	Lowkya Vuppu	Anjana Ouseph, Sai Anurag Vellanki, Nagarjun Reddy Mora, Chandra Sai Chalicheemala

Pulse from Github :

Excluding merges, 5 authors have pushed 29 commits to main and 44 commits to all branches. On main, 28 files have changed and there have been 15,549 additions and 3,881 deletions.

Code frequency stats from Git hub is as below :