

Project Phase III

Group 16

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 III of this project, includes the implementation of a command-line program to get counts of maps, distinct row labels, and distinct column labels, the modification of the bigDB and bigT classes to support multiple storage types, the implementation of various storage types using B-trees to index row and column labels, the modification of the batchInsert program to insert maps based on storage type and output disk page statistics, the implementation of a mapInsert program to insert individual maps. Also we aim to implement two types of join, they are sort merge join and cartesian join.

Problem Statement

In this phase, we implemented the functionalities that were asked to do in the phase III document provided to us. The major functionalities include sorting operator, joining operations, Map insert and get counts implementation.

Introduction

In the project phase II, we extend Minibase from a relational database management system into a DBMS with batch insert and query functionality that is similar to Bigtable. In this phase, we continue to develop this project by adding new features to increase its functionality.

The project phase III, entails changing the relational database management system MiniBase to produce a distributed database system that simulates various aspects of the DB on a single system. The project calls for the implementation of a number of programs, including the batchInsert program being modified, the mapInsert program being implemented, and the getCounts command-line program. The project also calls for the implementation of a BigT join operator, which creates a stream of maps with matching rows depending on predefined criteria. The project's objective is to build a useful database system with features similar to a Bigtable-like DBMS.

Assumptions

1. In row join User also passes the column filter value in COLUMNFILTER according to which we join each of the two tables.

Implementation and Proposed Solution

getCounts

Command: getCounts table_name NUMBUF

This command line invocation would display the number of maps, number of distinct row labels and number of distinct column labels using at most NUMBUF.

BigTable is a distributed database whose data content is scattered in different locations. In phase-3 we assume each heap file is an individual scattered file working from a

specific location. so when getCounts is called you have to get the count from the overall big table .i.e all 5 heap files and display the cumulative sum.

We implemented this by creating a new class called GetCounts, which would be initialized when the user chooses this option from the menu.

Getting count of maps :

To get this count, we iterate over all the heap files of the big table and get the total sum of all the maps in all 5 heap files.

Getting count of distinct row labels :

For this, we start a Stream on the table (combined with 5 heap files) of order type 1 so that it is ordered on row labels. We go over each map and compare the row label with the next and increment the count if it is different else just continue.

Getting count of distinct column labels :

For this, we start a Stream on the table (combined with 5 heap files) of order type 2 so that it is ordered on column labels. We go over each map and compare the column label with the next and increment the count if it is different else just continue.

Finally, we print the above 3 counts along with the read-and-write stats.

<Add a screenshot of the cmd here >

Changes to bigDB and bigT

BigTable is a distributed database which means that a single table is distributed across different locations as chunks. These parts should be always in sync and work together to address user queries or updates on the DB. We simulate this with these chunks of DB on a single system. We are assuming the data persisted into DB is distributed across 5 different heap files and all these heap files are in sync to address user actions.

We modified the bigDB class such that the constructor does not take type as input. In other words, in the same bigDB database, there may be data stored according to different storage types.

We modified the storage types to be the following as defined in the phase III specifications document.

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
Type 5	One btree to index row label and value

In all and all, we are maintaining 5 heap files one for each type defined above.

Changes to batch insert

We made slight changes to the batch insert functionality from the phase II implementation. The command to invoke it didn't change but the internal implementation has changed.

In the same big table, different maps may be stored according to different storage types, based on the batch in which they were inserted. We ensure that there are at most three maps with the same row and column labels, but different timestamps, in the entire big table – irrespective of which batch they were inserted in and which storage type they are subjected to.

To maintain at most three maps for the same row and column labels, we changed the logic of functionality. We are now maintaining an index on all the data across the 5 heap files which are stored in the 0th index of the index files in the bigT class. This index is on the column label and row label (it can be maintained on the row and column label too.).

When the batch insert command is invoked, we simply insert the data into the table i.e the respective heap file, the respective index file and the 0th index file that we are now maintaining. Once all the maps are inserted, we now call a method in bigT to delete the maps to maintain at most 3 maps of different time stamps for a row label and column label. The logic of this method is that it calls a map index scan on the index we are maintaining and goes over the maps and marks the maps that are to be deleted and we later go over them and delete them.

Through this, we maintain statistics of records inserted and total non-duplicate records in the table across heap files which we output along with the read-and-write stats to the console.

mapInsert

In this phase, we implemented a new command **mapInsert** which can be invoked by this cmd.

mapinsert RL CL VAL TS TYPE BIGTABLENAME NUMBUF

where BIGTABLENAME are strings and TYPE is an integer (between 1 and 5), RL row label, CL column label, VAL value, and TS timestamp.

To implement this, we created a new class called MapInsert which gets invoked on choosing this from the menu in the console. We check if the DB exists and the table exists. If not we create them. Post that we are inserting the map into the table and then we check if it is creating any duplicates and delete the map that is doing that.

Query

In this phase 3 we modified the query program little bit compared to phase 2. Here the command line invocation is without TYPE as below:

**query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER
VALUEFILTER NUMBUF**

The query functionality now fetches the records from all storage types. In batchinsert, when we insert a new heapfile we append the TYPE along with the file name into the database. So when we are querying, we don't need to specify the TYPE parameter again. It automatically fetches the file from the database if it exists along with its type.

RowJoin

Implemented a BigT join operator which joins two bigtables based on the column values from each of the tables and stores the resulting output bigtable into a new user defined bigtable.

We also implemented a command-line program join as shown below:

**rowjoin BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER JOINTYPE
NUMBUF**

Here the user passes the inner table name as BTNAME1 and outer table name as BTNAME2 and the output bigtable name which needs to store the resultant joined tables is passed by the user as OUTBTNAME.

User also passes the column filter value in COLUMNFILTER according to which we join each of the two tables.

JOINTYPE can be 1 or 2. If the user passes the value as 1, then we implement sort merge join and if it is 2 then we implement Nested loop join.

Minibase will use at most NUMBUF buffer pages to run the query.

The output is a stream of maps corresponding to a BigT consisting of the maps of the matching

rows based on the given conditions, such that

1. Two rows match only if they have the same column and the most recent values for the two

columns are the same.

2. The resulting rowlabel is the concatenation of the two input rowlabels, separated with a “.”

Sort Merge Join

The first type of join we implemented is Sort-Merge join.

Sort Merge Join is a database join algorithm used to combine two sorted relations or tables based on a join condition. It is commonly used in Relational Database Management Systems (RDBMS) to join large tables efficiently.

In a Sort Merge Join, both input relations are first sorted based on their join key, which is the attribute used to match the rows between the two tables. Then, the sorted relations are scanned sequentially in parallel, and matching rows from each relation are combined into a single result row. The algorithm uses two main steps: the sorting phase and the merging phase.

It has a best-case scenario cost of $M \log M + N \log N + M + N$, i. cost to sort R + cost to sort S + $([R] + [S])$ where the number of duplicates is small and can fit into a page. If we have a large number of duplicates the cost could go up to $M \log M + N \log N + M * N$ i.

Disadvantage of SM Join – it is a blocking operator. Because we have to obtain all the results from the lower level of the query plan, sort them and only then we can give the results to the upper level of the query plan.

If the results we get from the lower level of the query plan are already sorted the cost of SM Join would be even lesser.

Cartesian join

This method first generates an empty heapfile with the supplied outBigTName and, if one already exists, deletes its contents. The procedure then retrieves the Map objects from the leftStream and rightStream and stores them in the corresponding ArrayLists designated as outerRelation and innerRelation.

The method creates a new bigt object with one heap file page and returns it if either of these ArrayLists is empty. If not, the procedure builds a fresh bigt object with a single heap file page and joins the two ArrayLists in a nested loop to produce the result set.

Row sort operator

The name of the source table, the name of the output table, the name of the sorting column, and the number of buffer pages are all inputs for the class RowSort. The class's run() method executes the actual sorting algorithm.

The technique operates by splitting the source table into two heap files using a stream, one of which contains all rows sorted by most recent timestamp and the other of which only contains rows that meet a criteria on a given column. The algorithm then uses a sort iterator to sort the second heap file, inserting the sorted rows into the result table according to the chosen column.

Each distinct row label is added to the heap file using the createHeap() method, which also accepts a stream and a heap file. In order to add the appropriate rows to the result table, the dumpDefault() method checks the row labels in the two heap files. For choosing the rows with a particular row label, the getConditionalExpression() method provides a conditional expression. The command line for row sort operator is follows:

rowsort INBTNAME OUTBTNAME COLUMNNAME NUMBUF

OUTPUTS

1. Batch Insert

```
----- 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 -----
1
FORMAT : batchinsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
batchinsert src/test_data1.csv 1 first 500
Starting to read from the data file : src/test_data1.csv
Index type : 1
Table name : first_1
Number of buffers : 500
Replacer: Clock

TOTAL RECORDS READ FROM THE FILE : 10000
TOTAL NON DUPLICATE RECORDS : 6639
READ COUNT : 125
WRITE COUNT : 2
```


2. Query

```
----- BigTable Tests -----  
2  
FORMAT : query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF  
query first 1 Australia Zebra * 250  
[Australia Zebra 43652 ] -> 4159  
[Australia Zebra 92673 ] -> 7514  
RECORD COUNT : 2  
READ COUNT : 0  
WRITE COUNT : 0
```

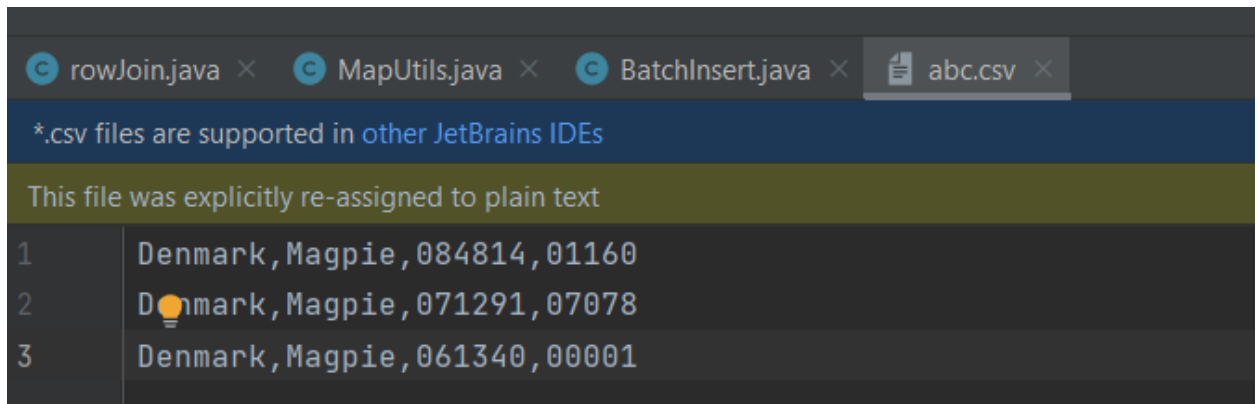
3. Map insert

```
----- BigTable Tests -----  
3  
FORMAT : mapinsert RL CL VAL TS TYPE BIGTABLENAME NUMBUF  
mapinsert dbms impl 24 1234 1 first 250  
TOTAL RECORDS READ FROM THE FILE : 1  
TOTAL NON DUPLICATE RECORDS : 6642  
READ COUNT : 0  
WRITE COUNT : 0  
----- 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 -----  
2  
FORMAT : query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF  
query first 1 dbms * * 250  
[dbms impl 1234 ] -> 24  
RECORD COUNT : 1  
READ COUNT : 0  
WRITE COUNT : 0
```

4. RowJoin

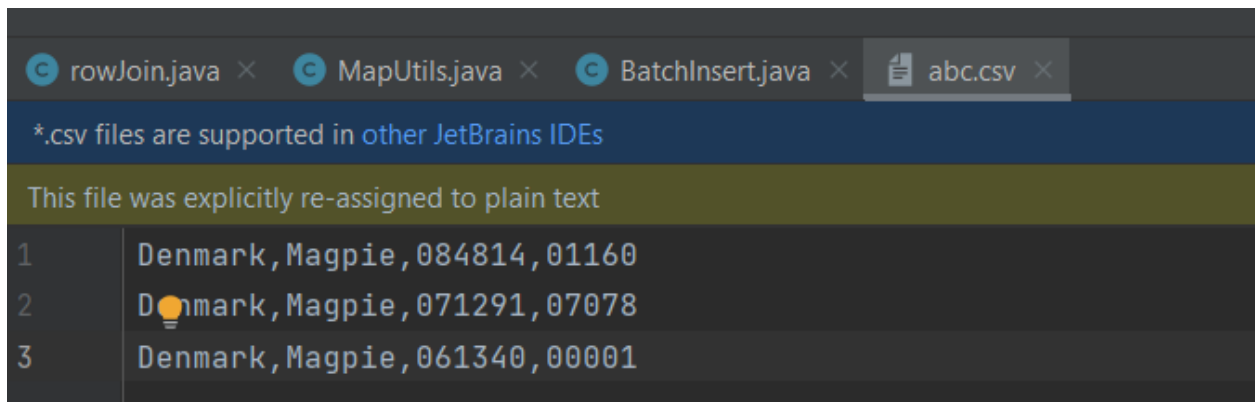
1. Sort Merge Join

For the below table “First”



1	Denmark,Magpie,084814,01160
2	Dnmark,Magpie,071291,07078
3	Denmark,Magpie,061340,00001

For the below table “Second”



1	Denmark,Magpie,084814,01160
2	Dnmark,Magpie,071291,07078
3	Denmark,Magpie,061340,00001

a) For the CLI : **rowjoin first second third Magpie 1 250**

We get the below result:

```
----- BigTable Tests -----
4
FORMAT : ROWJOIN BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER JOINTYPE NUMBUF
rowjoin first second third Magpie 1 250
.....You have chosen SortMergeJoin.....
.....Implementing SortMergeJoin.....

Query results =>
[Denmark:Denmark Magpie 61340 ] -> 1
[Denmark:Denmark Magpie 71291 ] -> 7078
[Denmark:Denmark Magpie 84814 ] -> 1160
RECORD COUNT : 3
READ COUNT : 1
WRITE COUNT : 0
```

b) For the CLI **rowjoin first second fourth Zebra 1 250** (here the column filter value is Zebra which doesn't exist in both tables so both tables won't be sorted and the tables won't be merged based on common column value as desired so we get an empty result)

```
----- BigTable Tests -----
4
FORMAT : ROWJOIN BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER JOINTYPE NUMBUF
rowjoin first second fourth Zebra 1 250
.....You have chosen SortMergeJoin.....
.....Implementing SortMergeJoin.....

Query results =>
RECORD COUNT : 0
READ COUNT : 0
WRITE COUNT : 0
```

- c) For the CLI **rowjoin anjana lowkya output Zebra 1 250** (here the two tables “anjana” and “lowkya” doesn’t exist, as we are trying to do rowjoin on two non-existing tables we expect an empty result)

```
----- BigTable Tests -----
4
FORMAT : ROWJOIN BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER JOINTYPE NUMBUF
rowjoin anjana lowkya output Zebra 1 250
.....You have chosen SortMergeJoin.....
.....Implementing SortMergeJoin.....

Query results =>
RECORD COUNT : 0
READ COUNT : 2
WRITE COUNT : 0
```

2. Cartesian Join

Data for below tables first and second are :

For *First* table

data1.csv		data2.csv	
1	Australia,Alligator,061340,00001		
2	Australia,Alligator,043215,00002		
3	Sweden,Alligator,010163,56788		

For *Second* table

	data1.csv	data2.csv
1		Denmark,Alligator,061344,00001
2		Arizona,Alligator,025215,00002
3		Kerala,Alligator,10163,00003

Cartesian join for *First* and *Second* tables:

```
FORMAT : ROWJOIN BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER JOINTYPE NUMBUF
rowjoin first second third Alligator 2 800
.....You have chosen CartesianJoin.....
.....Implementing CartesianJoin.....

Query results =>
[Australia:Arizona Alligator:Alligator 43215 ] -> 2
[Australia:Arizona Alligator:Alligator 61340 ] -> 1
[Australia:Denmark Alligator:Alligator 43215 ] -> 2
[Australia:Denmark Alligator:Alligator 61340 ] -> 1
[Australia:Kerala Alligator:Alligator 43215 ] -> 2
[Australia:Kerala Alligator:Alligator 61340 ] -> 1
[Sweden:Arizona Alligator:Alligator 10163 ] -> 56788
[Sweden:Denmark Alligator:Alligator 10163 ] -> 56788
[Sweden:Kerala Alligator:Alligator 10163 ] -> 56788
RECORD COUNT : 9
READ COUNT : 1
WRITE COUNT : 0
```

5. Row Sort:

First performed Batch Insert for table *one* with the test data provided in Ed Discussions - test_data1.csv

```
----- 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 -----
5
FORMAT: rowsort INBTNAME OUTBTNAME COLUMNNAME NUMBUF
rowsort one two Giraffe 800
NUMBER OF MAPS IN OUTPUT BIGTABLE: 6639
READ COUNT : 935
WRITE COUNT : 1396
```

Performed Query operation on the outBigTable *two* from rowsort

```
----- BigTable Tests -----
2
FORMAT : query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
query two 1 * Giraffe * 800
[Australia Giraffe 4730 ] -> 7660
[Australia Giraffe 39742 ] -> 3628
[Australia Giraffe 58338 ] -> 10000
[COSTA_RIC Giraffe 82688 ] -> 7465
[COSTA_RIC Giraffe 84358 ] -> 8535
[COSTA_RIC Giraffe 87662 ] -> 4579
[CZECH_REP Giraffe 26823 ] -> 1337
[CZECH_REP Giraffe 30988 ] -> 8488
[CZECH_REP Giraffe 82224 ] -> 2002
[Canada Giraffe 9859 ] -> 2984
[Canada Giraffe 25093 ] -> 1210
[Canada Giraffe 88295 ] -> 4813
[Colombia Giraffe 1544 ] -> 1589
[Croatia Giraffe 59217 ] -> 7468
[Croatia Giraffe 75110 ] -> 8302
```

6. GetCounts:

```
----- 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 -----
6
FORMAT : getCounts BIGTABLENAME NUMBUF
getCounts one 900
Table name : one
Number of buffers : 900
Map count : 6639
Distinct Row Label : 50
Distinct Col Label : 50
READ COUNT : 1868
```

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 the src folder to start the application. NOTE : test file is present in the code src folder named as test_data1.csv.

Conclusions

In this project, we were assigned with creating a DBMS that is similar to Bigtable utilizing the MiniBase components. By adding a number of new features and tweaking some already-existing ones, we were able to achieve this.

Throughout the course of the project, we learnt about the difficulties involved in setting up a distributed database system, such as managing various storage formats and ensuring consistency across numerous heap files. Additionally, we developed our skills working with MiniBase and adding Java-based database functionalities. Overall, we are pleased with the project's results and think we have effectively created a Bigtable-like database management system utilizing MiniBase as the base.

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
getCounts	Lowkya	
Making it distributed	Lowkya	
Modify the batchInsert program	Lowkya	
Implement a mapInsert program	Lowkya	
Modify the query program	Anjana	

RowJoin class	Anjana	
RowJoin type 1	Anjana	
RowJoin type 2	Anurag	Chandra
cmd for row join	Anjana	
external rowSort operator	Nagarjun	Chandra
cmd for row sort	Nagarjun	
Report	Lowkya, Chandra	Anurag, Nagarjun, Anjana

Code frequency stats from Github is as below :

