Minibase Implementation

CSE 510 Project Report – Phase 2 Group 21

Dustin Howarth
Arizona State University School of Computing and Augmented Intelligence
Arizona State University
Tempe, United States dghowart@asu.edu

Aaaron Yu
Arizona State University School of Computing and Augmented Intelligence
Arizona State University
Tempe, United States aaronyu@asu.edu

Pin-Hsin Kuo
Arizona State University School of Computing and Augmented Intelligence
Arizona State University
Tempe, United States pkuo12@asu.edu

Madhav Rajesh
Arizona State University School of Computing and Augmented Intelligence
Arizona State University
Tempe, United States mrajesh3@asu.edu

Daljit Bhalla
Arizona State University School of Computing and Augmented Intelligence
Arizona State University
Tempe, United States dbhalla@asu.edu

***Abstract*—Bigtable databases, such as those used by Google, are able to efficiently handle huge amounts of data [1]. This data is stored in order via timestamps and allows for more efficient querying [1]. These are becoming more popular in use and can be more efficient than classic database management systems (DBMS) depending on their use. A popular free and open source DBMS that is used is Minibase. Minibase however, uses a relational database management system (RDBMS), and as such we cannot take advantage of the Bigtable system. However, as we show in this report, by building off of the open source Minibase system, we can extend its functionality to create a quasi Bigtable DBMS while still maintaining and using the Minibase code foundation. This allows for users and developers with Minibase experience the ability to more easily implement and use a Bigtable DBMS. Additionally, five different indexing and clustering strategies have been created and analyzed so that users of this system can quantitatively determine the best indexing strategies to use based on the type of queries and batch insertions they are performing.**

***Keywords*—Database Management System (DBMS), Relational Database Management System (RDBMS), Bigtable, Minibase, Maps, Tuples, Indexing, Btree.**

I. INTRODUCTION

A. Terminology

Maps in the context of our database system is a data structure that contains four elements in the form of a row label (String), column label (String), timestamp (integer), and value (String). A tuple, of which we are replacing, is a variable length data structure that can have as many variables or values as the programmer wants. A Bigtable DBMS system is composed of maps, while a classic RDBMS is composed of tuples. A query is a prompt or program call that gets data from the database based on the query call. A scan is a sequential check, either for a record, or a specific index file. We use index files to more efficiently allow for queries and scans and we store these index files in binary trees (btrees). Batch insertions are large scale insertions of records into the database.

B. Goal Description

Modern databases are often required and expected to store huge amounts of data. An example of this would be Google, which stores huge amounts of user data as well as website information, and more. The result of this is that scalability and efficiency of queries within databases has become incredibly important and often even the focus of database design. In this regard, traditional RDBMS, such as Amazon's SimpleDB or Minibase, are inefficient and less scalable when compared to newer DBMS such as Google's Bigtable [1].

The goal of this project therefore, was to create a DBMS system that emulates a Bigtable DBMS system, using the Minibase system and its code as a base to build off of. A Bigtable-like DBMS system uses fixed size maps to store data. Using this type of DBMS allows for far more scalability then traditional RDBMS's These fixed size maps include a row label, column label, and timestamp in order to index the data [1]. The Minibase therefore had to be changed and updated to handle this new structure, as Minibase uses variable sized tuples to store and index data [2]. As a result, the end product that we wanted was to create this Bigtable DBMS and have it perform basic database functionality (query, batch insert, etc.) while maintaining good response times and efficient memory usage, with the advantages and potential scalability of a Bigtable database.

C. Assumptions

In this project we have assumed that the Minibase foundation we are building on will perform as expected. We assume that the users of our implementation are familiar with Minibase itself and have knowledge and experience working with or using databases. We also assume that the users are familiar with Java and are able to run it on their system, as Minibase and our implementation is written in Java. Our implementation is also assumed to only run as instructed/intended as per our project document, as since it is built on top of Minibase any functionality or testing outside of the project document can result in undefined or unknown behavior.

II. Description Of Solution

The first and most important extension of the Minibase code document is the usage of maps instead of tuples. Tuples in Minibase allow for a variable amount of values (so long as the size of the tuple does not exceed the threshold) which allows the database more flexibility in what data it can create and how it can use those tuples in its algorithms. The drawback of this is that any access or usage of the tuples requires additional processing by algorithms as the size and types of the tuple elements are unknown. Maps, on the other hand, have a defined structure and therefore their structure will always be known and utilized.

To create these maps, we create a new class called Map.java in a new package, called BigT (this will hold the new BigT classes). Each Map, as discussed earlier, has a row label, a column label, a timestamp, and a value. The labels and timestamp can all be used for indexing (of which we have five different types that we will discuss later), and the map has a set header that is set by the setHdr() function. All of the tuples across the entire Minibase code base are then replaced by maps. This requires refactoring of variables as well as redesigning of the algorithms that previously used the tuples, as the structure and usage of them

differs a lot in many of the algorithms. Heap files and the iterators that use maps in particular required a lot of refactoring and work to change as they use a lot of maps and different algorithms to store and traverse them.

Some example files that were changed are Sort (converted to MapSort), SortMerge TupleUtils (converted to MapUtils), FileScan, PredEval, Projection, HeapFile, and HFPage (among many others). MapUtils divides the map data into four cases for comparison. CompareMaptoMap() directly extracts the specified key(), that is, the string or value corresponding to the field. Additionally, CompareMapsOnOrderType() and CompareMapsOnInsertType() are also created in MapUtils to perform these comparison tasks.

Within the new BigT package we also added the BigT.java file. This file is the overarching control file that creates any and all of the heap and index files that represent the database. It also contains and allows for five different index and clustering strategies that we use in the BigT database. The five index strategies are as follows: a btree to index row labels, a btree to index column labels, two btrees (one with a combined key of the column and row label and another with the timestamp), two btrees (one with a combined key of the row label and value string and another with the timestamp), and finally two btrees (one with a combined key of the column label and value string and another with the timestamp). Each of these index strategies was chosen in order to give users the ability to choose what keys and combination of keys they wanted indexed as well as to determine what the best index and clustering strategies were based on the type of operations being performed (as well as what users would want optimized).

The first strategy only indexes the row labels. This was chosen as row labels can be unique or the same as other labels and as such can be good to use for single btree indexing. This allows indexing to group maps with the same row labels, at the cost of potentially having many maps in the same group. The second strategy only indexes the column labels for the same reasons. The third strategy involves using a combined key of the row and column labels for the first btree and another btree with the timestamp as the final key. This strategy was chosen because it combines both the row and column labels into a single key (allowing for more unique keys and less matches) while the timestamp tree can be used to further narrow or select a record, as there can only ever be three or less records with the same row and column labels. The fourth strategy was selected as having a combined keys of the row label and value strings, along with another btree that indexes the timestamps. This strategy allows for better querying of the values of records within the databases, while still allowing for row labels to further searches. The fifth strategy indexes column label and value as a combined key, as well as indexing timestamps. This strategy is similar to the fourth, but we wanted to test the performance of combining the value with row labels and column labels.

Our implementation then, based on the query and type of indexing the current database is using, performs index scans and file scans in order to find matching data, along with other iterators (such as sort) in order to sort the data into the proper order. These functions are from the command line in the cmdline package. This package creates the batchInsert and Query programs that can be used on the command line to create, insert, and query records from the database. This is done using all of the aforementioned files along with buffer and disk management and represents the basic operations of the database. The disk management is headed by the main control file bigDB.java and it creates the Bigt objects and is able to run different operations on the database. The buffer manager handles the buffer usage and allows for page reading and writing.

III. Interface Specifications

The main interface and overarching design of this project comes from the BigT package. The BigT package has four main java classes: bigT, Map, Stream, and RowSort. The main class that is instantiated and used to create the database is the bigT class. This class, via the constructor, takes in the name of the database, as well as the type. The type of database corresponds to the index/clustering strategy that the database will use as described in this paper's "Description of Solution" section (section 2). This creates the main database that the system will use (and you can create multiple) and using this object you can call numerous functions to create and manage the database. This also allows for users to choose the best indexing/clustering strategy for their needs, allowing them to optimize their query response times (an important need in huge databases).

You can get the number of rows, columns, and maps within the database via getters and can insert maps via the insertMap() function (takes a byte array that has the data of the map), insert an index file via an MID parameter (Map id), delete an index file using an MID parameter, and open a Stream object with different filters via the openStream() function. These map functions form the core of the Bigtable data structure, as they have a fixed header with a fixed number of attributes and values. Using this structure of records (along with the different access, index, and sort functions) allows for more scalability (one of the focal points of big databases).

The openStream() function then ties into and uses the Stream class, which is the main class that is used to open and run queries from. This class takes in an order type, one of five values that determines the ordering of the values. The ordering values are as follows: 1- order by row label, then column label, then timestamp, 2- order by column label, then row label, then timestamp, 3- order by row label, then timestamp, 4- order by column label, then timestamp, 5- order by timestamp. The next parameters are then a row filter, a column filter, and a value filter. These filters allow for queries to search for exact labels/timestamps, ranges of labels/values, or (if set equal to Null) match all labels/values.

The main programs that are used in tests and on the command line are the query and batchInsert programs. Query will initialize and open a stream to get the maps, with the parameters of the name of the big table, the type of big table, the ordering type, the row filter, the column filter, the value filter, and the number of buffer pages to use. This allows for custom filters and using the Bigtable system we designed, very fast query results even in a large database. The batchInsert program takes in the data filename that is being inserted, the type of Bigtable, and the Bigtable name. This program allows for a large number of insertions through a single call and allows for better scalability in the database.

IV. System Requirements And Execution Instructions

The system we tested our implementation on was a Windows machine through an IDE (specifically Intelliji). There are no hardware requirements to run this code, provided that the desktop or system has enough RAM and CPU resources to run modern desktop applications. There are two main tests that we have provided users. The first is a test file called IndexStratTest.java. This test file creates five different databases, each with their own indexing and clustering strategy, and then performs queries (the same ones on each of the databases) in order to determine their total runtime. Doing this allows us to compare and determine the best indexing and clustering strategies for this dataset (as explained in the Conclusion section). The second test driver that we created is a manual command line/user input driven program that allows users to call the batchinsert and query programs.

The batch insert program is run by typing “batchinsert DATAFILENAME TYPE BIGTABLENAME” into the command line, where DATAFILENAME is the name of the data file (csv) where the records are being read from, TYPE is the type of indexing and clustering strategy to use (1-5), and BIGTABLENAME is the name of the big table to create. This inserts a large number of records into the new database. The query program is run by typing “query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF” into the command line, where BIGTABLENAME is the name of the big table that is being queried, TYPE is the type of indexing and clustering strategy to use (1-5), ORDERTYPE is the type of ordering to give the results in (1-5), ROWFILTER is the filter to filter the row labels, COLUMNFILTER is the filter to filter the column labels, VALUEFILTER to filter the filter the value Strings, and NUMBUF which is the number of buffer pages.

V. Related Work

There is a paper by S. Ramanathan, S. Goel and S. Alagumalai, that we have used throughout this report in both our research and for reference that goes into detail on the differences, advantages, and disadvantages of both RDBMS and Big Table DBMS, in the context of cloud databases [1]. The paper heavily focuses on Amazon’s SimpleDB (a RDBMS) and Google’s Big Table DBMS for the analysis [1]. In this, the major takeaways of the paper are that the Big Table DBMS has a very real and large advantage when it comes to scalability, usability, and response time from queries [1]. The significant disadvantage as compared to RDBMS

is that it does not support SQL, and most importantly that it limits the expressiveness of your queries [1].

Another study that was conducted compared Google's Bigtable, Amazon's Dynamo Database (a newer DBMS that Amazon uses for its scalability), and Apache's Cassandra Database [3]. These three databases are all used to handle large datasets that are ever expanding and all of them do not use SQL (NoSQL) [3]. The main advantage of not using SQL and a traditional RDBMS is that they are far more scalable and efficient when handling huge swaths of data [3]. Amazon's Dynamo Database and Cassandra are unique in that they are a peer to peer distributed database, instead of a traditional server based database [3]. Cassandra is also unique in that it is also open source and built off of the designs of both Bigtable and Dynamo, though it is more similar to Bigtable [3]. In comparison with one another, Bigtable still has the best scalability and very good consistency [3]. However, with Cassandra being open source, it represents a good base and potential source of comparison for our database, as well as could be used in future work or updates to Minibase and our implementation [3].

VI. Conclusions

Our implementation was successful in creating a Bigtable Esque DBMS in some regards. Our implementation is able to create new databases of records and perform batch inserts of records (such as with the given example data). Our implementation also uses five different indexing and clustering strategies depending on the given parameters. All five strategies allow for batch insertions with varying degrees of runtimes for each. In our results we found that, after running the batch insertion multiple times, the different strategies all ran in about the same time, despite the type of indexing and clustering strategy chosen.

This is interesting because different indexing strategies create different amounts of btrees with different combined keys (all of which we expected to add additional runtime) but our results showed that this was not the case, and that users can choose whatever indexing strategy they want, with each taking about 44000 - 45000 milliseconds (44 - 45 seconds) to create and insert the given test data into the database.

However, we did have trouble getting the query functionality to work properly, as we can internally show with debug print calls that the maps are found when queried.

However, when the sort functionality is called/utilized, errors occur and the query fails. This is due to some internal issues within the sort functionality, where the maps are not being properly created or passed to functions as the code tries to create a negative sized array (as the length of the String data is not correct). Overall, our implementation still has issues in some regards (query functionality). We found that our implementation had a lot of integration issues and ran into issues with version control as well. Given more time, we would be able to sort out all the remaining integration issues and fix the query functionality.

References

[1] S. Ramanathan, S. Goel and S. Alagumalai, "Comparison of Cloud database: Amazon's SimpleDB and Google's Bigtable," 2011 International Conference on Recent Trends in Information Systems, Kolkata, India, 2011, pp. 165-168, doi: 10.1109/ReTIS.2011.6146861.

[2] *R. Ramakrishnan. "The Minibase Home Page". minibase/minibase.html. https://research.cs.wisc.edu/coral/minibase/minibase.html (accessed Feb. 5, 2023).*

[3] S. Kalid, A. Syed, A. Mohammad and M. N. Halgamuge, "Big-data NoSQL databases: A comparison and analysis of "Big-Table", "DynamoDB", and "Cassandra"," 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), Beijing, China, 2017, pp. 89-93, doi: 10.1109/ICBDA.2017.8078782.

VII. Appendix

Roles of Group Members:

Aaron Yu - BigT package (bigt.java, Map.java, Stream.java), project report, misc. bug fixes

Dustin Howarth - I created the IndexStratTest. This creates 5 databases, one of each index strategy type, populates them with the given data via a batch insert, and prints out the time it takes to batch insert the records. It also prints out the number of reads, writes, and unique attributes in each database. I created the MID class to replace RID and changed all of the RID usage throughout the tree to use MID. I changed the

Heap Package by converting the package to use maps and MID instead of tuples and RID. I updated the Iterator Package (most of the classes) to use maps and MID and updated the functions/algorithms to work with and use the new format/data type. I refactored code and worked on integration as there were a lot of differences in the trees so we ended up having to add back and rechange/commit code. I worked on solving bugs such as map errors that caused incorrect map creation, fixed exception issues, fixed incorrect size of value strings within maps, fixed issues within the test class, and more. I also wrote the vast majority of the report document.

Pin-Hsin Kuo - MapUtils, MapSort, bug fixes, integration, project report.

Madhav Rajesh - Disk Manager package, Iterator Package, Cmdline package (batchinsert and query), initial github setup.

Daljit Bhalla - Disk Manager package, Iterator package, Cmdline package (batchinsert & query).