CSE 510 - Database Management System Implementation

Spring 2020
Phase III
Group Number 12

Group Members

Abhilasha Mandal
Krisha Mehta
Sai Madhuri Molleti
Sanika Shinde
Yash Kotadia
Yash Sarvaiya

Abstract

The report describes the implementation of a Bigtable-like database management system(DBMS) on top of the Minibase database system. While Minibase is a relational DBMS with a record as its basic element, we modify and extend this system to support maps as a fundamental unit. Through this project, we also leverage the advantages that Bigtable offers in terms of flexibility, scalability, and sparseness. A Map in Bigtable consists of a row label, column label, timestamp, and value. The proposed solution provides the user with the functionality of indexing over row value, column value, timestamp, or a combination of these values. The report explains the goal of this project, the different components of the new system developed as well, how these components interact with each other as well as the system's performance. Furthermore, we also implement join functionality that is supported in all relational database management systems.

Keywords

Bigtable, Map, Minibase, B+ Tree Index, Heapfile, FileScan, IndexScan

Table of Contents

1. Introduction 3
2. Solution Description 5
   - 2.1. Components 6
   - 2.2. Results and Analysis 16
   - 2.3. Query Optimization (Stream) 18
3. Interface Specifications 21
   - 3.1. BigT 21
   - 3.2. Btree 26
   - 3.3. Disk Manager 31
   - 3.4. Global 33
   - 3.5. Iterator 36
   - 3.6. getCounts 40
   - 3.7. Batch Insert 40
   - 3.8. Map Insert 40
   - 3.9. RowSort 41
   - 3.10. RowJoin 41
   - 3.11. Query 41
   - 3.12. Buffer Manager 41
   - 3.13. Index 42
4. System Requirements/Installation and Execution Instructions 44
   - 4.1. System Requirements 44
   - 4.2. Instructions to Get Started 44
   - 4.3. Points to Note 44
5. Related Work 45
6. Conclusion 48
7. Bibliography 49
8. Appendix 50

Introduction

Phase I of the project focussed on studying the modules of Minibase, a relational DBMS. Building on that, this phase of the project involves building a Bigtable-like DBMS[4] on top of Minibase. While Minibase being a relational DBMS has higher predictability, Bigtable has its own advantages. It is easy to store structured as well as semi-structured data in bigtable. It also supports flexibility and sparseness. In this report we first explain our proposed solution and design choices we have made while encountering different challenges. We describe the different packages of the system and how the methods function and interact with other methods. Later, we state the system requirements and instructions to run this project. Lastly, we delineate the research papers we studied to get a better understanding of the motivation and need for developing such a system.

Terminology

- Bigtable - Collection of 5 heapfiles, each with a different storage type and index. It stores data in the form of maps.
- Heapfile - It is a type of unordered file organization.
- RowJoin - It replicates the join functionality of relational database management systems.
- RowSort - It sorts the rows of the Bigtable based on a column label.
- BigStream - It creates a stream of maps on the entire Bigtable in sorted order.
- SORTPGNUM - It is the number of buffer frames allocated for sorting.
- Versions - They correspond to maps with identical row label and column label but different values and timestamp.
- Versioning - Functionality of the DBMS to maintain the most recent 3 versions in the system.
- BatchInsert - An operation used for bulk insertion of maps in the bigtable.
- MapInsert - An operation used to insert a single map in the bigtable.

Goal description (problem specification)

The goal of this project is to develop a database like bigtable using the modules in Minibase. While Minibase is a relational DBMS, this new system resembles bigtable and consists of maps instead of the classic records. This system will enable 5 types of indexing strategies. The aim is for the system to have a better performance than relational DBMS. However, we also leverage useful functionalities provided by relational DBMS like joins. We use the number of disk reads and writes as a metric to evaluate the performance of our system.

Assumptions

Listed below are the assumptions we have made during the development of this project:

1. Max column label and row label size has been set to 50 characters.
2. Sort assumes a max map size, the only variable attribute in our map is value which we assume to have a maximum size of 20 bytes.
3. Maximum elements in the heaps used by sort are set to 500.
4. Database size is set to 20,000 pages.
5. Number of buffer frames to sort the records in each individual heeap file of a BigTable is set to 50 to allow sorting on 25,000 maps in a single heapfile.
6. In case of row/column/value filter of type range, it assumed that there is no space eg. [A,Y]
7. In the test data provided, we ignore the first three bytes of the file as they contain the utf8-encoding bytes of the file.

Solution Description

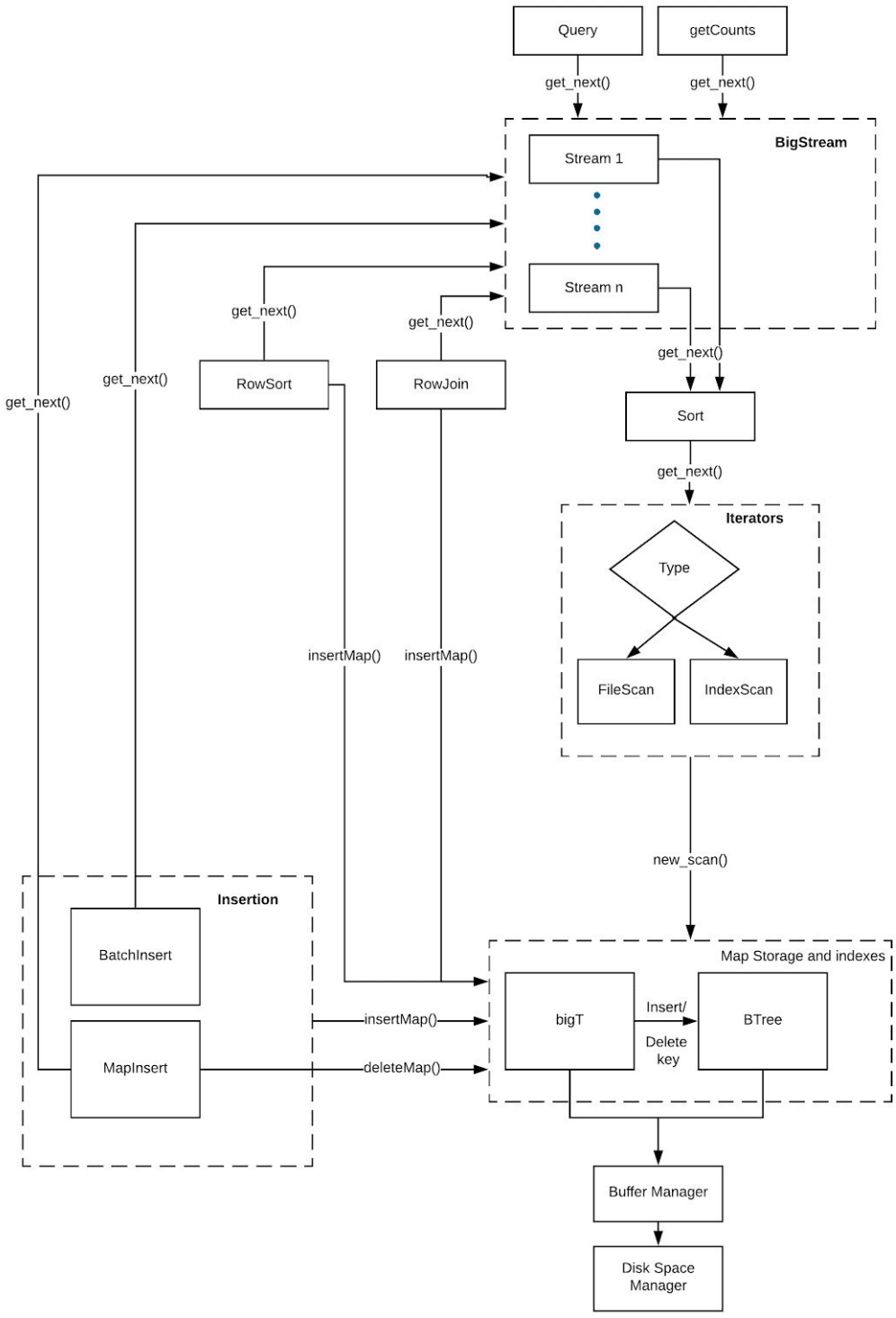


Figure 1. Architecture of Proposed Solution

Figure 1 illustrates an overview of the major components and their interfaces in the proposed solution. This section is divided into two parts: (1) The first part explains the design choices that were made for each of these components for minimizing the disk accesses. (2) The second part discusses the results of the experiments conducted using the proposed solution.

Components

1. Disk Space Manager

In phase 3 the disk space manager no longer manages the indexes, however, it still handles the index file creation or deletion. Since the BigTable now stores maps according to 5 different storage schemes, we maintain 5 different heapfiles for storing these maps. These storage schemes and their indexes have a one-to-one relation in the file directory(except for type 1). Each heapfile has its corresponding indexes. These indexes are created or deleted upon creation/deletion of the heapfiles.

The disk manager still has a counter for counting all the disk reads and writes which is printed and reset on closing the database. The next part of this section uses these results for analysis.

2. Map Storage and Organization

In phase 2 implementation we constructed BigT which stored maps belonging to specific index types. In this phase however, the BigT now encompasses 5 heap files which store the maps according to the specified storage scheme. This is achieved by maintaining a separate heapfile for each of the BigT storage schemes as well as a corresponding index file.

In phase 2 the disk space manager was responsible for accessing the indices, however, since in phase 3 a single bigtable is supposed to access multiple indexes, the responsibility of pinning the header pages of these indexes is relegated to the bigtable itself.

The versioning of maps by the current definition of BigTable is now more complex since it has to account for maps stored in different storage schemes(heapfiles). In the previous version of the project, the versioning was handled by the BigTable upon map insertion. However, this scheme was very expensive and is avoided in this phase by relegating the versioning to Batch Insert and Map Insert. During batch insertion, the versioning is handled by sorting and the entire data is rearranged. Map inserts handle the implicit deletions caused due to versioning by deleting the older map regardless of the heapfile it is physically present in. Apart from insertions, RowSort and RowJoin operations lead to

creation of a new BigTable, however, based on the nature of the operation we do not explicitly require to handle versioning.

Similar to phase 2, the directory pages of the heapfile based bigTs maintain the number of maps in each of the data pages. This facilitates quickly counting the total number of maps stored in the heapfile. Thus for the getCounts operation we can simply calculate the sum of map counts for each of the heapfiles deploying different storage schemes.

3. Indexes

According to project specifications, a single BigTable encompasses data stored according to different storage and indexing schemes. Since the data in this phase is stored in a sorted order, our unclustered indexes of phase 2 become clustered indexes. This is extremely beneficial for range queries. For an unclustered index each matching map would correspond to a disk access, whereas this is not the case for a clustered index.

4. Iterators

Iterators in Minibase provide a "get_next()" interface to query the heap files. The core utility of an iterator is to return the maps that satisfy the provided constraints in an efficient manner. Our project solution uses two types of iterators File Scan and Index Scan. While both the iterators simply provide a get_next() interface for querying, depending on the usage case, one would be more efficient over the other. The two types of iterators are further discussed as follows:

File Scan: It essentially opens a new scan on the heap file based bigT and scans all the records in the bigT sequentially. It also takes the row, column and value filters as input. The get_next() interface keeps calling the get_next() operation upon a scan and returns the maps obtained if they satisfy the filters.

Index Scan: Currently only unclustered indexes are used in the proposed solution. The advantage of using index scan is that it can efficiently position the scan pointer in the leaf pages. Starting from this position, the leaf pages are read and if the key satisfies the given constraints it returns the map. The scanning stops as soon as the index encounters a key that doesn't satisfy the constraints. An important observation is that while indexes are very efficient at finding the MID of any map, the Stream component expects a map from the Index Scan. Retrieving each map in such a scheme would require a disk access. Thus, using index scan for queries that don't significantly reduce the subset of maps that satisfy the constraints would significantly increase the cost.

For most purposes we do not directly access file scan or index scan. The Stream class provides a wrapper around these scans and optimally chooses between the two.

5. Sort

We have used the external disk based sorting provided by the minibase. This will sort the maps in the bigT depending on the orderType which is given as input to the sort. One caveat of the minibase sorting is that it requires fixed sized entries to sort. Thus, we pass this fixed size as input to the sort which corresponds to the maximum allowed size of a map. An arbitrary number of runs is then generated. For each run, a temporary file is created. Depending on the number of maps to be sorted, the number of these temporary files is increased dynamically. We play with two parameters, the SORTPGNUM, which is the number of buffers available in memory for the sort to use, and max_elements_in_heap which corresponds to the number of elements in the heap which will be present in a temp file and in memory at any point. SORTPGNUM has been set to 50 meaning that there are 50 buffers available in memory to sort approx. 25,000 maps in two passes. The max_elements_in_heap is set to 500. We found out that to sort around 5000 maps, 10 buffer pages are required. Giving very few buffer pages would lead to a low memory exception since sorting happens in only two passes of the tree created for external sorting. The maximum number of maps that will be present in main memory at any point is 500. Since we have arbitrarily chosen the maximum size of the map to be given as input to sort as approx. 1000 bytes, 500 maps would require 50 KB of RAM.

An error that we encountered and which was an error in the implementation of the existing minibase, was regarding these temporary files. During the sort, a spoof input buffer is used which reads the maps in the buffer using a scan. This scan however had not been closed due to which all pages were not getting freed, since some pages had their pin counts greater than 1. This was fixed by writing a close() method in this spoofIBuf which closes the scan and is called it in sort.close().

A parameter of type RowOrder has been added to the sort constructor which specifies whether sorting is to be done in ascending or descending order.

6. Insertions

BigTable insertions involve complex operations for maintaining the property of having at most 3 timestamps for one <row, column> pair. It is better to provide a wrapper around the functions of the BigTable.

A map can belong to any of the 5 storage types (and thus any of the 5 BigTables in the database) as mentioned in the specifications.

Following are two interfaces to insert maps into the BigTable.

a. Batch Insert

Batch Insert provides an interface to make bulk insertions into the respective BigTable. Since this implementation requires maintaining at most 3 timestamps for one <row, column> pair, all the existing Heapfiles need to be scanned to check for such matching maps and keep only 3 of them in the database. We read maps from the input csv file and insert them in a temporary Heapfile. We utilize the BigStream class functionalities to combine the streams created on existing Heapfiles, plus the temporary Heapfile (that stores all the new maps read from the input csv file) and to give out a sorted single BigStream. The BigStream is specifically created such that the output maps are ordered in row, then column and in decreasing order of timestamp to facilitate the selection of the maps that are to be stored in the database. Since the BigStream iterates over maps with decreasing order of timestamp, we keep the first three instances of matching maps and discard the remaining. Also, since the getNext() interface of the BigStream gives out maps in the order depending on the orderType using which BigStream is created and also holds the information of the Heapfile the map belongs to, it becomes easy to distribute the maps to their respective Heapfiles of the BigTable.

This implementation needs reordering of the maps in the 5 Heapfiles of the BigTable since the BigStream fetches maps ordered in row, column and in decreasing order of timestamp which is not the desired order for all storage types of the BigTable. Thus though the 5 Heapfiles contain maps of the correct category, these do require reordering in order to store them correctly on the disk. Thus, we create a Stream on each of these Heapfiles to fetch maps in the desired order. But minimizing the rework, this needs to be done only for Heapfiles with storing maps of storage types 3, 4 and 5.

b. Map Insert

Map Insert interface allows a single map to be inserted in the database according to its storage type. However, since this insertion is not direct and needs to check matching maps i.e. maps with the same <row, column> as of the input map, in order to maintain only 3 such maps, it incurs additional operations on the database, apart from simply inserting the map.

The major task of finding the matching maps, if any, is done using the BigStream class. We initialize a BigStream on this database by keeping the sort flag false, to reduce the sort operation overhead. Thus the order type passed as argument does not matter which is used only for sorting. This optimization intends to utilize the indexing only, except for storage type 1, to find the matching maps and there is no

need to perform sorting. Since we pass the row and column of the input map as filters to create this Stream, it gives out the matching maps. By iterating over these maps using the getNext(mid) method of BigStream, we find the map with the smallest timestamp. We locally store its mid and index of the Heapfile to which this map belongs using which the map is deleted. We chose to keep this deletion as a naïve deletion without map redistribution giving an advantage of reduced disk accesses at the time of deletion and also at the time of inserting a new map in future which can take the slot left by the deleted map. And yet, it still maintains the order in which the maps are required to be stored on the disk corresponding to the storage types.

In order to insert the input map in respective Heapfile and keep the storage order on disk intact, we need to reorder the existing maps and the input map combined. We create a new Heapfile of the storage type same as that of the input map to store these ordered maps. To fetch the maps already existing in the Heapfile under the respective storage type, we have used FileScan and not Stream keeping in mind that the maps are already stored in the order required, thereby initializing sequential scan. Thus using FileScan, we iterate over the maps, comparing each existing map with the input map to find the correct position where the input map can be inserted.

Another approach to implement insertion was to first insert the input map blindly in the Heapfile and then initialize a Stream on it so as to get a sorted sequence of maps. However, this incurs additional operation of sorting the already sorted maps.

7. Stream

The component Stream provides a getNext() interface to user queries. It contains the logic for query optimization that scans the BigTable in an efficient manner to return the maps in specified order.

The constructor for Stream takes rowFilter, columnFilter, valueFilter and orderType as input. The goal is to return maps that match these filters in the requested order. While leveraging the index would certainly help, since we are using unclustered index for non index-only queries, it would not be wise to use the index for each and every query. For example, using row index for a query with <rowFilter, columnFilter, valueFilter> as <“*”, “*”, “*”> would essentially require one disk access for every map present in the BigTable, thus, in this case it would be better to scan the entire database and sort it. Furthermore, for a single query we can only use one index at a time. However, the queries require ordering on multiple parameters. Thus, it becomes mandatory to use sort to maintain the order type. For example, if we have an index on row labels, the given

order type is on <row, column, timestamp>, and the query filter is <“rowFilter”, “columnFilter”, “valueFilter” >. After using the row index to filter the maps on the given rowFilter, we still have to sort the resulting maps based on the required orderType. Thus we observe that in most cases the indexes are only useful for reducing the number of maps to be manually checked against the filters and then sorted.
Based on the observation we just stated, we optimize the query by using index scan based on the filters. These cases are effectively shown in the table below.

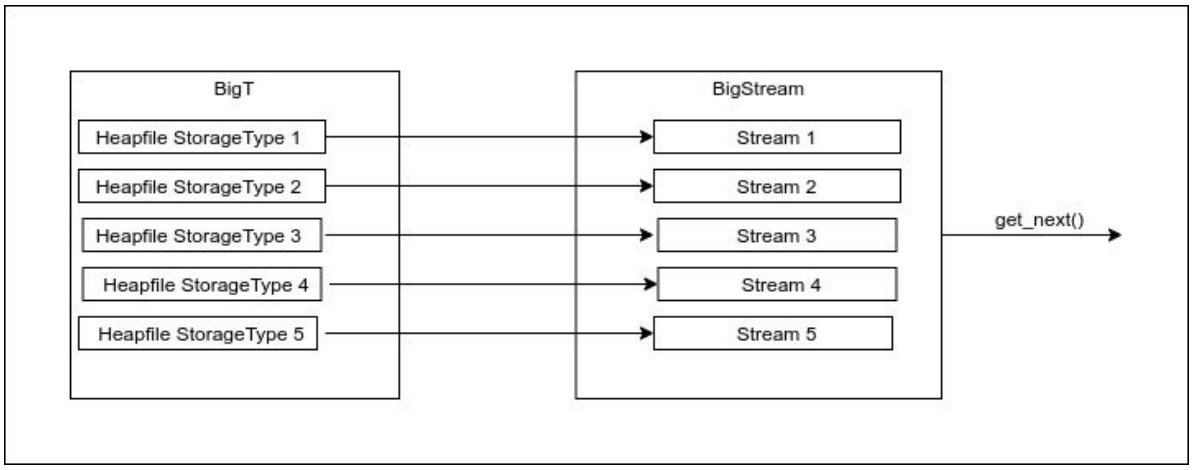
Index Type	Row Filter(RF)	Column Filter(CF)	Value Filter(VF)	Optimization Scheme
1	Any	Any	Any	Filescan
2	*	Any	Any	Filescan
2	Range, Equality	Any	Any	Index Scan <row> = <RF>
3	Any	*	Any	Filescan
3	Any	Range, Equality	Any	Index Scan <column> = <RF>
4	Any	*	Any	Filescan
4	Any	Range	Any	Index Scan <column,row> = <CF, *>
4	Any	Equality	Any	Index Scan <column,row> = <CF, RF>
5	*	Any	Any	Filescan
5	Range	Any	Any	Index Scan <row, value> = <RF, *>
5	Equality	Any	Any	Index Scan <row, value> = <RF, VF>

Table 1: Query Plan for various combination of filters and index types

As seen in the above table, the optimization scheme for most cases is self-explanatory. However, there are some interesting observations made in type 4 and type 5 which consist of composite keys. Consider the case where index type is 4, i.e, we have an unclustered index of <column, row> composite key. For the case where columnFilter is Equality, for all kinds of rowFilter we initialize an index scan <column, row> = <CF, RF>. However, when columnFilter is a range, then for all types of rowFilter we initialize

the index scan as <CF, *>. This is done because the current implementation of B+ Trees does not support *jumping* across leaf values. Thus, the index scan has to scan all the leaf values that do not satisfy the <columnFilter, rowFilter> in between two pairs that do satisfy these filters.

8. BigStream



The BigStream component is implemented to provide a getNext() interface similar to the getNext() of the Stream component, but this implementation handles maps of all the storageTypes in the database, instead of just 1 storageType. We have used the "K sorted streams" algorithm for creating one sorted stream from the 5 streams corresponding to each storageType.

The constructor takes Bigtable name, OrderType, rowFilter, columnFilter, valueFilter and a useSort flag as input. The implementation does not restrict the number of HeapFiles on which the BigStream can be created, thus making it robust and expandable.

The constructor initializes Streams on all the Heapfiles of the BigTable and stores the first map of these Streams in the array rMaps. While creating the Streams, the useSort flag is propagated further to the Stream component which decides whether to utilize Sort functionality or not, thus providing a flexible solution to reduce unnecessary overhead.

Initialisation of Streams ensures that the maps will be iterated over in a sorted order with the same filters and orderType. Thus, the k sorted streams algorithm applies correctly here.

The getNext() interface provides the output to be in ascending as well as descending order determined by the rOrder data member. We used a simple array rMap to hold the maps of the Streams that needed to be compared to find min/max map, which simplified the functioning of getNext(). Once the output map is found, we store the minInd value to

outInd, so as to know the Heapfile to which the output map belongs. This saved unnecessary scanning of all Heapfiles specifically for deletion operation. Thus the rMap array needs to be updated with every call to getNext() in order to iterate over the Streams correctly. Similarly we need to update the "done" flag too to identify the end of iteration of any of the Streams.

9. Row Join

This is a join operation between two bigTables, and aims to mimic the join operation in relational databases. This operation includes joining all the rows of BigTable1 with rows of BigTables2 (even though the internal representation of a BigTable is still maps), that match on the most recent value of a particular column filter.

We have used the Sort-Merge Join(SMJ) algorithm. Although both the bigTs(subset of bigTs) need to be sorted on values, this has many advantages for our implementation as compared to Block Nested Loop Join(BNLJ) and Index Nested Loop Join. Firstly, since we don't have an index on values, we cannot leverage the index nested loop join. We are however, leveraging other types of indexes as part of SMJ as will be discussed subsequently. The average case cost for the block nested loop join is also greater than that of the sort merge join. Moreover, the main advantage of the BNLJ is that it is a non-blocking operator which becomes beneficial for complex queries having multiple operators. However, our BigTable DBMS does not have complex queries and so the blocking SMJ operator does not cause any trouble on performance. Lastly, the expensive sorting operation performed for the SMJ, is performed only on a subset of maps(those that have a column label matching the column filter), and not on the whole bigTable. It uses BigStreams as discussed above.

Implementation of SMJ:

First two bigStreams are created, one on each bigTable to get all the maps having a column label matching the column filter. Two portions of the bigTable(Type 3 & Type 4) contain indexes on column labels and so getting these maps becomes relatively cheap. Once we have these maps, the most recents distinct maps are then filtered to temporary heapfiles which are then sorted on values, and later given as input to the SMJ. This is because for the SMJ, comparison happens on the most recent value, if there are multiple versions of the map. Now, once two maps(map1 & map2) are chosen as candidates to join, along with their join, we need to get all the maps(all the columns) having rowlabel same as that of map1 in bigTable1, and similarly for bigTable2. While doing this, the most recent three versions are also maintained. The required maps are then output to an output heapfile.

SMJ requires the two input streams to be sorted on values, however getting the most recent recent version of a map requires sorting the streams on rowlabel, columnlabel and

then timestamp. At a time, the streams can only be sorted on one criteria and thus we first sort on rowlabel, column label, timestamp to get the most recent versions of the maps, and then sort those maps on values to get to give as input to SMJ. Doing the reverse would be wasteful, because if two maps are selected as candidates to join, we would not know whether they are the most recent versions or not. So everytime two maps match on values, we would need to get all the versions of these two maps and check the equality of their most recent versions.

Another future optimization that can be done, is to alter the map definition to include a flag which indicates whether that map is the most recent of its version, or not. This can be done at the time of insertion, which makes it a cheap operation. This would remove the step of initial sorting done to get the most recent versions of the maps.

10. Row Sort (Implements Row Order)

Row sort allows the sorting of a BigTable based on the specified column filter. We take the names of the input and output BigTable as input parameters along with the column name over which the sorting is to be done. We initialize two BigStream called stream1 and stream2 based on order type 1 and apply the column filter on stream1. Along with this, we have also implemented additional row order support that enables the user to organize rows in ascending and descending order. Stream1 is used to determine the list of maps that satisfy the column filter. Once we get this stream, we compare it with stream2 and list down all the maps in stream2 having rows commons with that of stream1. These maps are added to the output BigTable. Rows of maps present in stream2 but not in stream1 are added to a temporary heapfile. In the end, they are appended to the list in the output BigTable. Hence, we obtain a BigTable with rows sorted based on a column label.

11. getCounts

This command line invocation counts the number of maps in the database, distinct row labels, and distinct column labels using at most NUMBUF buffers.

The directory pages of BigT store records of the DataPageInfo of each data file i.e. each HFPage. We have stored the number of maps that each data page stores and thus by iterating through the records of the directory pages, we have calculated the number of maps in the database without actually accessing the data page or retrieving the maps. Thus, this requires disk accesses just equal to the number of directory pages of the BigTable.

To find the distinct row labels or the column labels, we create a BigStream, with orderType 1 and 2 respectively, which gives out the maps ordered in row/column. Thus by simply comparing the row label/column label of the current map with that of the previous map, we calculate the distinct row labels and column labels. This

implementation leverages all the advantages that BigStream and thus Stream + Sort functionalities offer. Since this phase of the project has maps with 5 storage types unlike the previous phase, sorting maps over row/column was an economic approach to find distinct labels, if the count was not to be recorded at the time of insertion.

12. Query

The query component provides a wrapper around the BigStream component. While Stream encapsulates complex query optimization logic, BigStream combines these individual sorted streams on different storage schemes.

Results and Analysis:

RowSort:

The following query is used for reporting the results of RowSort. We have reported an ablation study on two parameters, i.e, RowOrder and the number of buffers.
java RowSort input output RowOrder Sheep NUMBUFF

#Buffer	750	1000	1500
ASC	R: 8886 W: 6861	R: 8853 W: 6854	R: 8814 W: 6850
DSC	R: 8956 W: 6903	R: 8885 W: 6889	R: 8838 W: 6887

We have set the constant SORTPGNUM to 50 buffers. Since there are 5 heapfiles in one bigTable and we are using 3 bigStreams (input bigTable, outputBigTable, temp BigTable), it would require at least around 750 buffers for the operation. This number can be reduced if SORTPGNUM is reduced. The average number of reads and writes remains the same because the number of maps in the input table is the same and sorting requires a fixed number of buffers.

BatchInsert:

The experiment for batch insert is designed as follows:

- Since the performance of our batch insert is now dependent upon the number of maps previously present in the BigTable, we test the BatchInsert operation on three states of the database.
- *State 1*: The BigTable is empty. We perform BatchInsert on storage type 1.
- *State 2*: The BigTable contains 10K maps in storage type 1. We perform BatchInsert on storage type 2.
- *State 3*: The BigTable contains 10K maps each, in storage type 1 and storage type 2 . We perform BatchInsert on storage type 3.
- An ablation study is also provided on the number of buffers used.

#Buffer	State 1 Insert 10K in type 1	State 2 Insert 10K in type 2	State 3 Insert 10K in type 3
500	R: 2838 W: 4216	R: 5745 W: 7211	R: 16974 W: 17790
750	R: 2803 W: 4210	R: 5716 W: 7206	R: 12449 W: 13507
1000	R: 2744 W: 4200	R: 5694 W: 7203	R: 11899 W: 12986

The above table depicts the disk access results that we obtained by varying 2 parameters, i.e, Number of Buffers available for insertion and the state of the BigTable i.e. number of maps already present. We analyze the results obtained by varying these two parameters as follows:
We have set the constant SORTPGNUM to 50 buffers. Since there are 5 heapfiles in one bigTable and we are using 3 bigStreams (input bigTable, outputBigTable, temp BigTable), it would require at least around 750 buffers for the operation. This number can be reduced if SORTPGNUM is reduced.

In state 1, the batchinsert operation sorts the input maps, removes duplicates and adds the maps to the heapfile depending on the storage type.
In state 2, since there are already existing maps in the bigTable, the batchinsert operation creates a sorted bigstream on these old maps as well as new maps to be inserted and then removes duplicates, before redistributing the maps to their corresponding heap files. The number of reads and writes thus increases because we need to maintain consistency of versions across the entire bigTable.
In state 3, since there are 20k maps already in the bigTable, the bigStream now needs to be created on a total of 30k maps and then remove duplicates and redistribute. The larger number of maps already existing explains the larger number of reads and writes.

Number of Buffers:
The number of reads and writes(in a particular column above) remains almost the same because sorting requires a fixed number of buffers(decided by SORTPGNUM), and number of maps (in a particular state/column) is the same.

MapInsert

The MapInsert operation is intended for inserting single maps into an in-use bigtable potentially after batch insertion has already been done. For the experiment to be reflective of the indexing capabilities of the BigTable, we first do BatchInsert of 5K maps into each of the storage schemes and then perform single map insertion. We also perform an ablation study on the number of buffers.

#Buffer	500	750	1000
	R: 1290 W: 852	R: 1287 W: 852	R: 1287 W: 852

The MapInsert operation uses an unsorted bigStream to get all the versions of the map which needs to be inserted and then inserts the map in its correct position in the heapfile corresponding to the storage type. Inserting the map at its correct position leverages the underlying sorting order and so the corresponding heapfile does not need to be resorted after insertion of the map. Due to this, the number of reads and writes remains the same.

Query:

Our current implementation of Query encapsulates the get_next() interface provided by BigStream to access an entire BigTable. Since a BigTable consists of maps stored according to different storage and indexing schemes the only variables for this experiment are the types of column filter and row filter (equality, range, *). We do performance analysis based on some example queries. For our experiment we have batch inserted 5K maps in each of the storage schemes. Since the value filter does not impact the optimization scheme, we pass value filter as * for all the example queries. Following are the example queries and their observed reads/writes.

1. *query yo 1 "Monaco" "Giraffe" "*" 500*
   Reads: 488, Writes: 14

2. *query yo 2 "Monaco" "[E,P]" "*" 500*
   Reads: 1190, Writes: 203

3. *query yo 3 "Monaco" "*" "*" 500*
   Reads: 1834, Writes: 29

4. *query yo 4 "[D,K]" "Ram" "*" 500*

Reads: 816, Writes: 103

5. *query yo 5 "[D,K]" "[E,P]" "*" 500*
   Reads: 1693, Writes: 467

6. *query yo 4 "[D,K]" "*" "*" 500*
   Reads: 2761, Writes: 744

7. *query yo 3 "*" "Goose" "*" 500*
   Reads: 1770, Writes: 38

8. *query yo 2 "*" "[A,D]" "*" 500*
   Reads: 2897, Writes: 879

9. *query yo 2 "*" "*" "*" 500*
   Reads: 6158, Writes: 3080

All the optimizations involved in the above queries are performed by Stream. The optimizations across each of the streams are done based on the storage and indexing scheme, and the filters as shown in Table 1.

getCounts:

Since each of the heap files in the current BigTable maintains the map count, map count of the entire BigTable is trivial. The current implementation uses sorting for calculating the number of distinct rows and columns. We test our implementation in a simple experiment where have 10K maps stored in each of storage type 1,2 and 3.
#Reads: 13653
#Writes: 6785
The reads can be attributed to Filescan of all heapfiles as well as sorting. The writes are completely because of the temporary heapfiles created during sorting.

RowJoin

State of tables:
Yo: Contains 10k maps stored as type 1
Yo2: Contains 10k maps stored as type 2

1. *rowjoin yo yo2 yoout1 Echinorhi 1500*
   Reads: 4344, Writes: 138

In this rowjoin, a single row in both tables is being joined. Initially two bigstreams are created to get the most recent versions of each map. These are then output to a temporary heapfile, over which a stream is created sorted on values. Now, once two maps have been selected as candidates to join, two temporary bigstreams are again created to get all the columns of the matched rows. Due to the creation of multiple bigstreams and streams, the number of reads and writes is high.

2. *rowjoin yo yo yoout2 Echinorhi 1500*
   Reads: 142034, Writes: 2110
   This is the case of a self join on a bigTable having 10k records. Since each map is joined with itself, to get all the columns for that row, we need to create a bigstream for every joined map. Potentially there can be 20k bigstreams created (depending on the number of distinct rowlabel, columnlabel pairs) and so the number of reads is very high.

Interface Specification

The implementation of this phase of the project involved the creation and modification of multiple files. In this section, we explain why and how we have modified the original Minibase database system to adapt to our problem description. Below, each package is described with respect to its function in the proposed solution.

1. BigT

a. DataPageInfo

It is the type of records stored on a directory page of the bigT. Each record of this type refers to a data page in the bigT. Along with a pointer to the page, it stores necessary information about the page such as the free space available and number of maps on the page. The format of the record is as follows:

availspace	**mapct**	**pageId**
4 bytes (int)	4 bytes (int)	4 bytes (int)

Constructors and Methods:

- ***public DataPageInfo():*** This creates a byte array of size 12 (4*3), and initializes the map count and available space variable to 0.
- ***public DataPageInfo(byte[] array):*** Initializes DataPageInfo from a byte array.
- ***public byte [] returnByteArray():*** It translates a tuple to the DataPageInfo object.
- ***public Tuple convertToTuple():*** Converts DataPageInfo object to a tuple, and returns a tuple.
- ***public void flushToTuple():*** Writes DataPageInfo's useful fields (availspace, mapct, pageId).

b. HFPage

Directory pages, as well as data pages in the database system, are hfpages when loaded in memory. However, while data pages use maps, directory pages use tuples. To ensure both these data structures are supported, we overloaded the functions.

c. Map

This file defines the class for a map in our BigTable. It provides various constructors to create a map. It provides functions to get and set row, column, timestamp, value as well as size. It also encapsulates the logic for calculating the size of the map since the size of a map is variable

It encodes an entire map in a byte array. The format of the data byte array is as follows:

map_length	**row_label**	**column_label**	**timestamp**	**value**
4 bytes	Fixed in global constants	Fixed in global constants	4 bytes	variable

d. Scan

This file defines a class to facilitate sequential access and retrieval of the maps stored in one heapfile of the BigTable. It contains information of bigTable being used, page ID of current directory page, the directory page itself (HFPage), data page Record ID (Since directory pages store records in form of tuples), actual page ID of data page, a copy of this data page, Map ID of the current map and Map ID of the outputted map.

Constructors and Methods:

- ***public Scan(bigT bgt):*** The constructor of Scan initializes the Scan object by subsequently making calls to init(HeapFile in BigTable) and firstDataPage() methods. It pins the first directory page and also identifies the first data page, thereby initializing corresponding data members.
- ***public Map getNext(MID mid):*** This function retrieves the map next to the map identified by input argument mid. It gives out the currently stored usermid as an maps output map and then sets this usermid by fetching the next map, thereby facilitating output map information when the next call to getNext() is made. This method also embeds a call to the nextDataPage() method when the nextUserStatus flag is false. This flag(initially true) monitors if the scan has iterated over all maps in a data page and thus the next data page needs to be fetched.
- ***public void closescan():*** This method resets all the data member values to their defaults and also unpins the directory page and the data page.

e. Stream

This file implements a class to retrieve maps from the bigTable in the orderType specified by the user. When the order type is:
1, then results are first ordered in row label, then column label, then time stamp.
2, then results are first ordered in column label, then row label, then time stamp.
3, then results are first ordered in row label, then time stamp.
4, then results are first ordered in column label, then time stamp.
5, then results are ordered in time stamp.
Constructors and Methods:

- ***public Stream(bigT bt, int orderType, int rOrder, String rowFilter, String columnFilter, String valueFilter, boolean useSort):*** The constructor, called only from the method openStream() of bigT class, utilizes Sort class functionalities along with FileScan and IndexScan class to create a stream of maps in the desired order (based on order type). It exploits FileScan or IndexScan to create a scan on the maps, based on the type of the bigDB database and the row/column/value filters. IndexScan is used to optimize the query only if the filter is not "*" in the case of an indexed database (type 2, 3, 4, 5). If the database is created with no indexing, FileScan is passed to Sort only if the useSort flag is true.
- ***private CondExpr[] GetConditionalExpression(String filter, int dbType):*** This method formulates the row, column or value filter in a conditional expression to be fed to IndexScan. It supports the 2 types of filters that can be given by the user.
  Filter 1 – Range Search – For this filter, it uses minibase attribute operator aopGE and aopLE. Since it is a range search, 3 conditional expressions are defined with the third one being null. Expression 1 defines the respective fields corresponding to aopGE operator (Greater than equal to). Expression 2 defines respective fields corresponding to aopLE operator (Less than equal to).
  Filter 2 – Equality Search – For this filter, it uses operator aopEQ. 2 conditional expressions are defined with the second one being null. Expression one defines respective fields corresponding to aopEQ operator (Equal to).
- ***private CondExpr[] getCondExpr(String filter1, String filter2):*** This method formulates the row, column or value filter in a conditional expression to be fed to IndexScan. A combination of filter 1 and filter 2 decides which conditional expression is to be created. This method uses aopGE operator (Greater than equal to) with aopLE operator (Less than equal to), to create the expression for both range search and equality search.

- ***public Map getNext():*** This method fetches the next maps in the stream by calling a get_next call to the sort data member initialized in the constructor or scan data member depending on whether UseSort is set to true.
- ***public void closestream():*** This method closes the stream by subsequently calling close on the sort/scan data member.

f. Tuple

This class defines a tuple as it is used in directory pages. It holds the tuple length, tuple offset, which is its start position, the number of fields and an array of the field offsets. The data is stored as a byte array. The maximum size allowed to a tuple is the Minibase page size.

A tuple can be created by calling the default constructor which creates a byte array and sets the offset to zero and the length to maximum. It can also be done by passing values for the fields mentioned, passing the size, or from another tuple. There is also a method to copy a second tuple into the current tuple position, given that their lengths match. Finally, there are a few methods to access specific fields.

g. bigT

This bigT implementation is heapfile directory-based. It consists of directory pages and data pages. The directory pages contain tuples of records and data pages contain maps. All directory pages are in a doubly-linked list of pages, each directory entry points to a single data page, which contains the actual maps.

interface Filetype: This decides whether the bigT is ordinary or temporary (used in sorting).

private HFPage _newDatapage(DataPageInfo dpinfo): Gets a new datapage from the buffer manager and initializes dpinfo.

private boolean _findDataPage(MID mid, PageId dirPageId, HFPage dirpage, PageId dataPageId, HFPage datapage, RID rpDataPageRid): Internal bigT function (used in getMap and updateMap). It returns a pinned directory page and a pinned data page of the specified user map(mid) and true if the map is found.

public bigT(String name): Constructor that initializes the bigT with its name. A null name initializes a temporary bigT. If the name already denotes a file, the file is opened; otherwise, a new empty file is created.

public int getMapCnt(): This returns the number of maps in the bigT.

public MID insertMap(byte[] mapPtr): Inserts a map into heap file, return its Mid.

public boolean deleteMap(MID mid): Deletes a map from the bigT with given mid.

public boolean updateMap(MID mid, Map newmap): Updates the specified map in the bigT.

public Map getMap(MID mid): Read a map from file, returning pointer and length.

public Stream openStream(int orderType, int rOrder, String rowFilter, String columnFilter, String valueFilter, boolean useSort): Initializes a stream of maps depending on the filters.

public Scan openScan(): Initializes a sequential scan of maps in the bigT.

public void deletebigT(): Deletes the bigT from the database.

private void insertMapIndex(MID mid, byte[] mapPtr): Called in insertMap, used to update the BTreeFile, when a new map is inserted.

public String get_fileName(): Returns the name of the bigT.

h. BigStream

This class provides functions to create a stream on Heapfiles in BigTable. These streams can be made to be sorted or unsorted.

public BigStream(String bigtableName, int orderType, int rOrder, String rowFilter, String columnFilter, String valueFilter): creates a sorted stream on the bigT

public BigStream(String bigtableName, int orderType, int rOrder, String rowFilter, String columnFilter, String valueFilter, boolean useSort): creates a sorted or unsorted stream on the bigT depending on the value of useSort passed

public BigStream(String[] bigtableNames, int orderType, int rOrder, String rowFilter, String columnFilter, String valueFilter): creates separate sorted streams on the bigTs whose names are passed

public BigStream(String[] bigtableNames, int orderType, int rOrder, String rowFilter, String columnFilter, String valueFilter, boolean useSort): creates sorted or unsorted streams on the bigTs depending on the value of useSort passed

public Map getNext(): returns the next map in the stream

public Map getNext(MID mid): returns the next map in the stream that mid belongs to

public void close(): closes all the streams

2. Btree

The package Btree encapsulates all the files that are required to create and maintain a B+ Tree for maintaining unclustered indices of the Big Table. The code from the minibase has been modified for our purpose. The major modifications have been made to support composite keys. BigTable type 4 and type 5 use (String, String) composite keys. Following is the list of files contained in the btree package.

a. IndexFile

This is an abstract class that defines the basic insert and delete interfaces provided by an index file. This class is extended by BTreeFile which defines a B+ Tree index.

b. BTreeFile

This file defines the B+ Tree indices in the BigTable. It extends IndexFile. It contains the constructors to create a new B+ Tree or open an existing B+ Tree. It calls the disk manager to create new B+ Tree files or delete existing ones. It maintains the header page of the B+ Tree.

It defines the insert method to insert a pair into the B+ tree given a key and a map ID. It checks whether the key being inserted is of the same type as the BTree file. It traces along with the BTree Index pages and inserts the <key, MID> pair in the appropriate leaf page.

The delete method deletes the given <key, MID> pair from the B+ Tree. It finds the appropriate leaf page to make the deletion. The full delete method of deletion does merging or redistribution while the naive delete method simply just deletes the pair.

public BTreeFile(String filename): It takes the filename as an input parameter. Based on that, an index with that filename is opened.

public BTreeFile(String filename, int keytype, int keysize, int delete_fashion): Based on the filename, if it exists, it opens it. Otherwise it creates it.

public void close( ): Closes the B+ tree file and Unpins the header page.

public void destroyFile( ): It destroys the entire B+ tree file.

public void insert(KeyClass key, RID rid): It inserts a record with the given key and rid.

public boolean Delete(KeyClass key, RID rid): It deletes leaf entry given its <key, rid> pair.

BTLeafPage findRunStart (KeyClass lo_key, RID startrid): It takes the left-most occurrence of ‘lo-key’, going all the way left if lo_key is null as an input

parameter. The other input parameter is startid that will return the first rid =< lo_key. It returns a BTLeafPage instance which is pinned.

public BTFileScan new_scan(KeyClass lo_key, KeyClass hi_key): It creates a scan with the given keys based on the cases mentioned below:

- lo_key = null, hi_key = null - scan the whole index
- lo_key = null, hi_key!= null - range scan from min to the hi_key
- lo_key!= null, hi_key = null - range scan from the lo_key to max
- lo_key!= null, hi_key!= null, lo_key = hi_key - exact match (might not unique)
- lo_key!= null, hi_key!= null, lo_key < hi_key - range scan from lo_key to hi_key

c. KeyClass

It is an abstract class for defining a Key that is inserted in the database. The key may be String, Integer or composite (String, String).

d. Integer Key

Defines the constructor, getter and setter for accessing the private integer variable.

e. String Key

Defines the constructor, getter and setter for accessing the private String variable. It extends KeyClass.

public String toString(): returns the key.

f. StringString Key

Defines the constructor, getter and setter for accessing the private (string, string) pair. This is mainly used for composite keys. It extends KeyClass.

public StringStringKey(String s1, String s2): This constructor takes the two strings as input and based on that assigns the value to two unique keys.

g. BT

It is a utility class that defines various static methods for facilitating some operations of the BTreeFile and debugging. The leaf pages and index pages of the BTree pages are uncoupled with the logic of the type of the key being stored. This is done by the BT class by defining the logic for comparing two keys of the same type. Thus, the logic for comparing the composite (String, String) key is defined in BT. It has functions for converting a <Key, Data> pair to a byte array and vice

versa. The debugging functionality is provided by methods to print the required sections of the B+ Tree.

public final static int keyCompare(KeyClass key1, KeyClass key2): It compares the two keys passed as input parameters. Returns negative if key1 is less than key2. Returns positive if key1 is bigger than key2.

protected final static int getKeyLength(KeyClass key): Returns the length of the given key.

protected final static int getDataLength(short pageType): It takes pageType (Whether NodeType is leaf or index) as input parameter. If node type is leaf, it returns the length as 8, if node type is index it returns the length as 4. Otherwise, it throws a NodeNotMatch exception.

protected final static int getKeyDataLength(KeyClass key, short pageType): Returns the sum of key length and data length.

protected final static KeyDataEntry getEntryFromBytes(byte[] from, int offset, int length, int keyType, short nodeType): It gets a KeyDataEntry from bytes array and position. Returns a KeyDataEntry object.

protected final static byte[] getBytesFromEntry(KeyDataEntry entry): It takes data entry as an input parameter and converts this keyDataEntry to a byte array with size equal to the size of (key,data).

public static void printPage(PageId pageno, int keyType): Mainly used for debugging, it takes the pageno and key type as an input parameter and prints the page out. The page is either BTIndexPage or BTLeafPage.

public static void printBTree(BTreeHeaderPage header): Also used for debugging. Takes the head page of the B+ tree file as an input parameter and prints out the B+ tree structure.

public static void printAllLeafPages(BTreeHeaderPage header): Used for debugging. Takes the head page of the B+ tree file as an input parameter and prints out all the leaf pages of the tree.

h. KeyDataEntry

KeyDataEntry encapsulates a Key and its corresponding data in the B+ Tree. Depending on whether the pair belongs to a leaf page or index page, the data member may be a page no. or a Map ID.

public KeyDataEntry(String key, PageId pageNo): Takes the key and page no as input parameters and encapsulates it in the B+ tree as an index page entry.

public KeyDataEntry(String key1, String key2, PageId pageNo): Takes two keys and the page no as input parameters and encapsulates it in the B+ tree as an index page entry that has a composite key.

public KeyDataEntry(Integer key, MID mid): Takes an integer key and MID as input parameters and encapsulates it in the B+ tree as a leaf page entry.

public KeyDataEntry(String key, MID mid): Takes a string key and MID as input parameters and encapsulates it in the B+ tree as a leaf page entry.

public KeyDataEntry(String key1, String key2, MID mid): Takes two string keys and MID as input parameters and encapsulates it in the B+ tree as a leaf page entry.

public boolean equals(KeyDataEntry entry): Takes the entry to check the key as an input parameter. Returns true if the key is equal to the entry. Returns false otherwise.

i. BTreeHeaderPage

It defines an interface to the header page of the B+ tree. It inherits the HFPage class. It provides access to the root of the B+ Tree. It also stores various properties of the B+ Tree like maximum key size, delete fashion and key type.

public BTreeHeaderPage(PageId pageno): It pins the page with pageno, and gets the corresponding SortedPage

public BTreeHeaderPage(Page page): It associates the SortedPage instance with the Page instance

public BTreeHeaderPage( ): Initiates a page, and associates the SortedPage instance with the Page instance

j. BTSortedPage

It extends the HFPage and defines the pages in the B+ tree. The records contained by these pages might be a <Key, PageID.> pair or <Key, MID> pair. It maintains the abstract records in a sorted order based on the key comparison interface provided by BT.

public BTSortedPage(PageId pageno, int keyType): It takes the pageno and key type as input parameters. It pins the page with pageno, and gets the corresponding SortedPage.

public BTSortedPage(Page page, int keyType): It associates the SortedPage instance with the Page instance.

protected RID insertRecord(KeyDataEntry entry): Performs a sorted insertion of a record on an record page. The records are sorted in increasing key order. Only the slot directory is rearranged. The data records remain in the same positions on the page.

public boolean deleteSortedRecord(RID rid): It deletes a record from a sorted record page. It also calls HFPage.compact_slot_dir() to compact the slot directory.

k. BTFileScan

BTFileScan implements a search/iterate interface to B+ tree index files (class BTreeFile). It derives from abstract base class IndexFileScan.
public KeyDataEntry get_next(): It iterates once (during a scan).
public void delete_current( ): It deletes the just scanned data entry.
void DestroyBTreeFileScan( ): It is a destructor that unpins a few pages if they are not unpinned already and does some cleaning work.

l. BTIndexPage

It defines an index page in the B+ tree. It extends the BTSortedPage to maintain the <Key, PageID> records in sorted order. It implements the logic for performing redistribution and merging. It also provides the interface for accessing the records on the page.

- ***public BTIndexPage(PageId pageno, int keyType)***: It pins the page and gets the corresponding BTIndexPage. It also sets the node type to NodeType.INDEX.
- ***public BTIndexPage(Page page, int keyType)***: It associates the current object with the Page instance passed and sets the node type as above.
- ***public BTIndexPage(int keyType)***: It creates a new page and associates it with the BTIndexPage. It also sets the node type as above.
- ***public RID insertKey(KeyClass key, PageId pageNo)***: It inserts a key into the index page and returns the RID of where it was inserted.
- ***RID deleteKey(KeyClass key)***: It deletes the key passed and returns it.
- ***PageId getPageNoByKey(KeyClass key)***: It returns the page number to be searched next.
- ***public KeyDataEntry getFirst(RID rid)***: It returns the first entry on the index page.
- ***public KeyDataEntry getNext (RID rid)***: It returns the next entry on the index page.
- ***int getSibling(KeyClass key, PageId pageNo)***: It returns -1 for a left sibling, 1 for a right sibling and 0 for no siblings.
- ***boolean adjustKey(KeyClass newKey, KeyClass oldKey)***: It replaces the specified old key with the new one and returns true if successful. If no such key is found, it returns false.

- ***KeyDataEntry findKeyData(KeyClass key)***: It returns the entry for a key if found, null otherwise.
- ***KeyClass findKey(KeyClass key)***: It returns the key if found, else, null.
- ***boolean redistribute(BTIndexPage indexPage, BTIndexPage parentIndexPage, int direction, KeyClass deletedKey)***: It redistributes the records after a key has been deleted and returns true if the redistribution was successful, false otherwise.

m. BTLeafPage

It defines a leaf page in the B+ tree. It extends the BTSortedPage to maintain the <Key, MID> records in sorted order. It implements the logic for performing redistribution and merging. It also provides the interface for accessing the records on the page.

public BTLeafPage(PageId pageno, int keyType): It pins the page based on the pageno and sets the to be NodeType.LEAF.

public BTLeafPage(Page page, int keyType): It associates the BTLeafPage instance with the Page instance, also it sets the type to be NodeType.LEAF.

public RID insertRecord(KeyClass key, RID dataRid): It involves two RIDS with two different meanings. The takes a **dataRID** as an input parameter along with the key. It inserts a key, rid value into the leaf node. This dataRID is the rid of the data record which is stored on the leaf page along with the key. It returns a **RID** of the inserted lead record data entry, i.e the <key, dataRID> pair.

public KeyDataEntry getFirst(RID rid): It takes the RID as an input parameter. It provides an iterator interface to the records on a BTLeafPage. It returns the first KeyDataEntry in the leaf page.

public KeyDataEntry getNext (RID rid): It takes the RID as an input parameter. It provides an iterator interface to the records on a BTLeafPage. It returns the next KeyDataEntry in the leaf page.

public KeyDataEntry getCurrent (RID rid): returns the current record in the iteration; it is like getNext except it does not advance the iterator.

public boolean delEntry (KeyDataEntry dEntry): Based on the KeyDataEntry passed, it deletes the data entry in the leaf page. Returns true if deleted, else returns false.

3. Disk Manager

a. bigDB

public void openDB(String fname): It opens the database with the given name.

public void openDB(String fname, int num_pgs): Overloaded method to open the database with given name and specified number of pages.

public BTreeFile initIndex(String bt_name, int type): Initializes the indices for all storage types.

public void closeDB( ): It closes the DB file.

public void DBDestroy( ): It destroys the database completely by removing the file that stores it.

public void read_page(PageId pageno, Page apage): It takes the page no and page object as input parameters and reads the contents of the specified page into a Page object.

public void write_page(PageId pageno, Page apage): It takes the page no and page object as input parameters and writes the content in a page object to the specified page.

public void allocate_page(PageId start_page_num): It allocates a set of pages where the run size is taken to be 1 by default. It gives back the page number of the first page of the allocated run with default run_size =1.

public void deallocate_page(PageId start_page_num, int run_size): It deallocates a set of pages starting at the specified page number with run size = 1.

public void add_file_entry(String fname, PageId start_page_num): It takes the file entry name and start of the page number of the file entry. It adds a file entry to the header pages.

public void delete_file_entry(String fname): It takes the file entry name as an input parameter and deletes the entry corresponding to that file from the header pages.

public PageId get_file_entry(String name): It takes the file entry name as an input parameter and gets the entry corresponding to the given file.

public void dump_space_map( ): A space map is a bitmap showing which pages of the database are currently allocated. The method prints out this space map.

private void pinPage(PageId pageno, Page page, boolean emptyPage): It takes the pageno, page and a boolean about whether the page is empty or not as input parameters and provides a shortcut to access the pinPage function in the buffer manager package.

private void unpinPage(PageId pageno, boolean dirty): It takes the page no as an input parameter. Along with it also takes a boolean as an input parameter that represents where the pinned page is dirty or not. It creates a shortcut to access the unpinPage function in the buffer manager package.

public DBHeaderPage(Page page, int pageusedbytes): It is the constructor for class DBHeaderPage and takes the page and number of bytes used on that page as input parameters.

b. Page

All the data in the BigTable is stored in pages. These pages encapsulate the byte array storing the data. It has the definition of the constructor, getter and setter for a page.

c. Pcounter

Pcounter is responsible for counting the number of reads and writes.

4. Global

a. AttrType

Attribute type lists the attributes available such as String, Integer, Real, Symbol and Null. To this list, we added the attribute type StringString to support the composite key(String, String) for Type 4 and Type 5 BigDB.

public String toString(): It is used to return the attribute type depending on whether it is a string, integer, real, null or StringString .

b. Convert

It is a utility class that provides static functions to read and write various data types from a byte array.

public static int getIntValue (int position, byte []data)*:* It takes a byte array and the position of the required integer as input parameters. It reads 4 bytes from the given byte array at the specified position and converts it to an integer.

public static short getShortValue (int position, byte []data): It takes a data byte array and the position of the required integer as input parameters. It reads 2 bytes from the given byte array at the specified position and converts it to a short integer.

public static String getStrValue (int position, byte []data, int length): It takes the position, data byte array and length as input parameters. It reads a string that has been encoded using a modified UTF-8 format from the given byte array at the specified position.

public static String[] getStrStrValue (int position, byte []data, int length): It takes the position, data byte array and length as inputs parameters. Two strings that have been encoded using a modified UTF-8 are read from the location pointed by the position.

public static char getCharValue (int position, byte []data): It takes position, data byte array and length as input parameters. It reads 2 bytes from the given byte array at the specified position and converts it to a character.

public static void setIntValue (int value, int position, byte []data): It takes integer value that has to be copied , position of data, a data byte array as inputs. This is used to update the given byte array with the value.

public static void setShortValue (short value, int position, byte []data) : It takes the short integer value that has to be copied , position of data, a data byte array as input parameters. This is used to update a short integer in the given byte array at the specified position.

public static void setStrValue (String value, int position, byte []data): It takes the string value that has to be copied , position of data, a data byte array as inputs. This is used to insert or update a string in the given byte array at the specified position.

public static void setCharValue (char value, int position, byte []data): It takes a character value that has to be copied , position of data, a data byte array as input parameters. This is used to update a character in the given byte array at the specified position.

c. GlobalConst

Global constant contains the row label size and column label size needed to define a map. Both the row label and column label are set to 30 bytes. The maximum array size and number of buffers is set to 50. The page size and buffer pool size is set to 1024 . The database size is 10000 , maximum transactions is 100 and shared memory size is 1000.

d. MID

It defines map Id (MID) for the database based on the page no and slot no of the map entry in a given page. Since row id and map id have similar functionality, MID extends RID. This is extremely useful in places like btree that uses tuples.

e. SystemDefs

This file passes the type to BigDB to create the database. It is also responsible for initializing buffer manager and disk manager. After initializing the disk manager, it can either create a new database or open an existing database. When provided with a database name, it first checks if it already exists. If the database exists, it simply opens it otherwise it creates a new database. Lastly, it safely closes the database once all the executions are completed. To do this, it first closes the index after which it flushes all the pages, Finally, it closes the database.

public void init(String dbname, String logname, int num_pgs, int maxlogsize, int bufpoolsize, String replacement_policy, int type): It takes the database,

logname, number of pages, ,maximum log size, buffer pool size, replacement policy of the buffer and the type as input parameters. It initializes the buffer manager based on the buffer pool size and replacement policy. It also initializes the bigDB based on the type. The database name is used to check if it already exists, otherwise a new new database is created.

public void close(): It is used to close the index, flush all the pages and then finally close the database.

f. AttrOperator

It's an enumeration class for attribute operators .

public String toString(): It compares the attribute values with the given values using the negated operator of the sub query and results in a string value.

g. IndexType

It defines the index types in the database. Our system has a Btree index i.e B_Index. Hash index has been defined for future expansion.

Public String toString(): It takes an integer as input and returns the index type. For input = 0 there is no index. For input = 1 it is btree indexed. For input = 2 it is hash indexed.

h. PageId

Defines the constructor of the class. Along with this, it creates a copy of the page id and writes the pid into a specified byte array at offset.

public void copyPageId (PageId pageno): It takes the pageno as an input parameter and makes a copy of the corresponding page.

public void writeToByteArray(byte [] ary, int offset): It takes byte array and offset as input and the page id is copied into the array at the location pointed by the offset.

public String toString(): It returns the string value of integer class by passing page id as parameter.

i. RID

It defines the record id for each entry based on its page id and slot no. Along with this, it creates a copy of the rid and writes the rid into a specified byte array at offset.

public void copyRid (RID rid): It takes the record id as the input and makes a copy of the given rid.

public void writeToByteArray(byte [] ary, int offset): It takes a byte array and offset as input and writes the rid at the offset location.
public boolean equals(RID rid): It takes a rid as input and compares *this* to the rid. It returns true if they are equal and false if not equal.

j. MapOrder

Encapsulates an integer that specifies the order in which a map should be arranged in. This includes ascending, descending and random. The logic is encoded in a file named TupleOrder.java in the codebase.
public String toString(): It returns order of maps depending on the integer value. If the value is 0 then it is ascending, 1 then it is descending, 2 then it is random.

5. Iterator

a. BigSortMerge

- ***public BigSortMerge(String[] bnames1, String[] bnames2, Stream s1, Stream s2, String outBigTableName):*** The constructor initializes all the data members of this class that implements an interface for the sort merge joins.
- ***public void performJoin():*** Performs row join operation over the inputBigTables and inserts the result into output bigtable.
- ***public void close():*** Closes the input streams and deletes temporary BigTables used.

b. File Scan

This is a subclass of Iterator used to scan all maps from a file that match a given filter condition. It has the following methods:

- ***public FileScan (String file_name, String rFilter, String cFilter, String vFilter):*** The constructor initializes the row filter, column filter and value filter. It creates an object of class bigT from the file passed and then sequentially scans the Heapfile of the BigTable.
- ***public Map get_next():*** It implements the abstract method of Iterator to get the next map from the resultant Scan object and, if it matches the filters, return it.
- ***public boolean checkFilters(String mapLabel, String filter):*** It checks if the filters match a particular map.
- ***public void close():*** This abstract method of Iterator is implemented to clean up by closing the Scan object.

c. Iterator

This is an abstract class to implement relational operators and access methods for maps. It has:

- ***public abstract Map get_next():*** It is an abstract method to get the next map in sequence that every subclass must implement.
- ***public void get_buffer_pages(int n_pages, PageId[] PageIds, byte[][][] bufs):*** This is a method to acquire buffer space.
- ***public void free_buffer_pages(int n_pages, PageId[] PageIds):*** This method frees buffer space.
- ***private void freePage(PageId pageno):*** It frees the buffer page whose PageID has been passed.
- ***private PageId newPage(Page page, int num):*** It allocates a run of new pages and returns the PageID of the first page.
- ***public abstract void close():*** This is an abstract method for cleaning up that all subclasses must implement.

d. MapUtils

This class has some methods that are useful while processing maps:

- ***public static int CompareMapWithMap(Map m1, Map m2, int orderType):*** It is a method to compare two maps on various combinations of map fields depending on the orderType.
- ***public static String Value(Map map, int map_fld_no):*** This returns the row label, column label or value, depending on the field number passed.
- ***public static void SetValue(Map value, Map map, int map_fld_no):*** It allows the second map to set a specified field of the first map.

e. OBuf

It is a class that creates an output buffer. It has these methods:

- ***public OBuf():*** The default constructor takes no arguments and does no initializations.
- ***public void init(byte[][][] bufs, int n_pages, int mSize, bigT temp_fd, boolean buffer):*** The init method initializes a temporary buffer to pages, the number of pages, the map size and a reference to a bigT.
- ***public Map Put(Map buf):*** It is used to write a map from the current buffer to the output buffer.
- ***public long flush():*** This method returns the number of maps written to the output buffer.

f. Sort

This class is used to sort a file in the specified order (ascending or descending), depending on the orderType. It extends Iterator to access maps.

- ***private int generate_runs(int max_elems):*** This method uses heap sort to generate sorted runs of those maps.
- ***private void setup_for_merge(int map_size, int n_R_runs):*** It creates a setup for merging those runs.
- ***private Map delete_min():*** This deletes the minimum value of all the runs and returns the map.
- ***private void MIN_VAL(Map lastElem):*** It is used to set each field of the last element to the minimum value of the appropriate type.
- ***private void MAX_VAL(Map lastElem):*** It is used to set each field of the last element to the maximum value of the appropriate type.
- ***public Sort(TupleOrder sort_order, Iterator am, int n_pages, int oType, int valueSize):*** The constructor initializes the Iterator and tries to acquire as much buffer space as specified in its parameters. It also creates and initializes an OBuf with an equal amount of space.
- ***public Map get_next():*** The abstract method of Iterator is implemented to return the next map in sorted order.
- ***public void close():*** This overrides Iterator.close() to clean up by closing the Iterator object, freeing the buffer pages and deleting temporary files.

g. IoBuf

It is used to create an I/O buffer. It has the following methods:

- ***public void init(byte bufs[][], int n_pages, int mSize, bigT temp_fd):*** Instantiates the data members.
- ***public void Put(Map buf):*** Writes a tuple to the output buffer buf
- ***public Map Get(Map buf):*** It gets a map from current buffer, pass reference buf to this method.
- ***public long flush():*** It returns the numbers of tuples written.
- ***public void reread():*** If WRITE_BUFFER is true, this method is called to switch to read buffer.

h. SpoofIbuf

It is used to create an I/O buffer. It has the following methods:

- ***public void SpoofIbuf():*** The constructor initiates the Scan object of the class to null.

- ***public void init(bigT fd, byte bufs[][][], int n_pages, int mSize, int Nmaps):*** It initializes a reference to a bigT, an I/O buffer, the number of pages in this buffer, the map size and the number of maps. Then, it initiates a sequential scan of the heapfile in BigTable and assigns it to the Scan object.
- ***public Map Get(Map buf):*** This gets a map from the current buffer and returns it.
- ***public boolean empty():*** It returns true if the buffer is empty; else, false.
- ***private int readin():*** This method returns the number of maps in the buffer.
- ***public void close():*** It closes the Scan object.

i. pnode

This class is used to describe a map. It has the number of the run to which the map belongs and a reference to the map. It has the following methods:

- ***public pnode():*** It sets the run number to zero and the map reference to null.
- ***public pnode(int runNum, Map m):*** It sets them to the passed values.

j. pnodePQ

It is an abstract class to implement a sorted binary tree with these methods:

- ***public pnodePQ():*** The constructor sets the element count to zero.
- ***public int length():*** It returns the total number of elements in the tree.
- ***public boolean empty():*** It checks if the tree is empty.
- ***abstract public void enq(pnode item):*** It must be implemented by every subclass to insert an element in the tree.
- ***abstract public pnode deq():*** It is to be implemented to delete an element.
- ***public int pnodeCMP(pnode a, pnode b, int ordertype):*** This is a method to compare two elements and return -1 if a is smaller, 0 if they're equal or 1 if a is bigger.
- ***public boolean pnodeEQ(pnode a, pnode b, int ordertype):*** This checks if two elements are equal.

k. pnodeSplayPQ

It extends pnodePQ and implements a sorted binary tree.

- ***public pnodeSplayPQ():*** The default constructor creates a root node and initializes the sort order to *ascending*.

- ***public pnodeSplayPQ(TupleOrder order, int ordertype):*** It creates the root and initializes the sort order and orderType with the passed values.
- ***public void enq(pnode item):*** This is to insert an element into the tree in the specified order.
- ***public pnode deq():*** dequeue methods are implemented.

6. getCounts

This program is a command line interface implemented to output the number of maps, distinct row labels and distinct column labels. Input arguments include - Bigtablename and number of buffers.

To output the number of maps, it uses the getMapCnt() method implemented by the BigT class. It initializes all the Heapfiles in the Bigtable using the BigTableName input argument. It creates a BigStream over the BigTable to iterate over the maps in the specified order to find distinct row labels and column labels.

To find distinct row labels - ordertype 1, i.e. ordered in row, column then timestamp, is used. To find distinct column labels - ordertype 2, i.e. ordered in column, row then timestamp, is used.

7. Batch Insert

This program intends to create a bigTable and insert map data from the input file.

Input arguments include – csv filename, type (database type), bigtablename, number of buffer pages.

It implements the globalconst interface to the instantiate database and opens the comma separated value (csv) file whose name is taken as the input from the command line. The type , bigtablename and numbuff inputs decide the name and the number of buffers used. The values are read using " ," comma as delimiter. A Buffered reader is used to read by buffering characters in the file till a null value is obtained. Each line in the csv file is stored as a map that has a row label, column label, value and timestamp. This bulk insertion takes into account that only 3 maps with the same <row, column> pair exist in the BigTable with the help of BigStream to scan the existing maps. The program prints the total number of disk read writes incurred, when closing sysdef.

8. MapInsert

It is a program created to insert a map into a bigT. The arguments passed are the row label, column label, value, time stamp, storage type, bigT name and number of frames allocated. If the map is a duplicate, the oldest of the 3 existing maps is deleted and the new map is inserted.

9. RowSort

This program takes as arguments the name of the bigTable to be sorted, the name of the output bigTable, the row order, the column to be sorted on and the number of buffer frames allocated. 2 BigStreams are created on the input bigT. The first stream has maps that match the column filter in sorted order and the second stream contains all maps in sorted order. Until the streams are exhausted, maps from both streams are fetched and their row labels are compared, so that every map belonging to a row that has the column is stored in the output bigT, sorted on the column. The other rows are stored in a temporary bigT and are added to the output bigT in the end.

10. RowJoin

This is a program to join two bigTables on a common column. It takes as arguments the names of the input bigTable and the output bigTable, the column filter and the number of frames. Two BigStreams are initialized on the 2 bigTables by passing the column filter and setting the orderType to 6. Then, 2 temporary bigTables are created to hold just the most recent maps from both BigStreams, and 2 Streams are created on them. Finally, a BigSortMerge is initiated to join the 2 Streams and the result is the output bigTable.

11. Query

This program intends to open an existing database if present, otherwise create one, and retrieve maps of the bigtable in the order taken as input (ordertype).

Input arguments include – bigtablename, ordertype (output order form of the maps retrieved), row filter, column filter, value filter and number of buffer pages.

The program opens the existing database by giving numPages as 0 to the SystemDefs interface. It retrieves the bigTable using the input bigtablename, creates a BigStream object using the filters as input. The getNext method on the stream object is used to retrieve the desired maps. It also prints the total number of disk read writes incurred during the retrieval, when closing sysdef.

12. Buffer Manager

This package implements the operations of a buffer manager in minibase through the classes below:

- ***public class BufMgr***: This class stores HashTable (class BufHashTbl), buffer pool, array of descriptors one per frame and replacer object. The constructor initializes these members according to the specifications inputted while instantiating the database. Conversely, an object of buffer manager is set in the replacer too (We have used Clock replacement policy).

- ***class BufHashTbl:*** This class implements a buffer hash table to keep a track of pages in the buffer pool and manages (insert, retrieve, delete) pages in the hash table. The table size is set to 20. This class holds a hash table entry description (class BufHTEntry) in an array, where each slot holds a linked list of entries. The hash function used is PID mod (table size). It implements the *method insert()* to update hash table entry by hashing page no PID. The *method lookup()* is implemented to retrieve the frame no corresponding to the page no. to be found in the hash table. The *method remove()* implements deletion of the page from the hash table.
- ***class BufHTEntry:*** This class implements a buffer hashtable entry description. It stores each entry for the buffer hash table, the page number and frame number for that page. It defines a pointer that points to the next hash table entry.
- ***class Clock extends Replacer:*** This class implements the clock replacement policy.
- ***class FrameDesc:*** This class stores run time information and the status of each page in the buffer pool. It handles per page stats - the page number in the file; dirty status; pin count; pin count change when pinning or unpinning a page through *methods pin()* and *unpin()*.
- ***abstract class Replacer:*** This is the super class for buffer pool replacement algorithm and describes which frame to be picked up for replacement.

13. Index

h. IndexScan

This is a class that is used to directly access a map with the key provided, and for selections and projections. Its methods are:

- ***public IndexScan(IndexType index, final String relName, final String indName, String rFilter, String cFilter, String vFilter, CondExpr selects[]):*** The constructor sets up the index scan by initializing the index type (Btree or hash), name of the input relation, name of the input index, row filter, column filter, value filter and the conditions to apply.
- ***public Map get_next():*** If the scan is index-only, it returns the key. Otherwise, it returns the next map that matches all filters.
- ***public void close():*** It cleans up by closing the index scan.
- ***private boolean checkFilters(String mapLabel, String filter):*** This checks if a map field matches the corresponding filter.

i. IndexUtils

This class opens an index scan based on selection conditions with these methods:

- ***public static IndexFileScan BTree_scan(CondExpr[] selects, IndexFile indFile):*** This method opens a BTFileScan on the IndexFile passed.
- ***private static KeyClass getValue(CondExpr cd, AttrType type, int choice):*** It returns the key value from the condition expression.

System requirements/installation and execution instructions

System Requirements:

- A Unix/Linux based system
- JDK version 1.8.0_242

Instructions to get started:

- Check if there exists a Makefile in each of the packages to be compiled.
- Update "JDKPATH" in these Makefile files.
- Run command - make db; to compile the code
- Delete any temporary database files that exist in /tmp/ directory, if any, to make sure clean run.
- Compile and run the queries - BatchInsert and query.
    - **getCounts syntax:**
      javac getCounts.java
      java getCounts BIGTABLENAME NUMBUF
    - **BatchInsert syntax:**
      javac BatchInsert.java
      java BatchInsert DATAFILENAME TYPE BIGTABLENAME NUMBUF
    - **MapInsert syntax:**
      javac MapInsert.java
      java MapInsert RL CL VAL TS TYPE BIGTABLENAME NUMBUF
    - **Query syntax:**
      javac query.java
      java query BIGTABLENAME ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF
    - **RowJoin syntax:**
      javac RowJoin.java
      java rowjoin BTNAME1 BTNAME2 OUTBTNAME COLUMNFILTER NUMBUF
    - **RowSort syntax:**
      javac RowSort.java
      java RowSort INBTNAME OUTBTNAME ROWORDER COLUMNNAME NUMBUF

Points to note:

- Each filter can be of below 3 types:
    - "*" - This has to be passed in inverted commas.
    - Single value
    - Range specified by two column separated values in square brackets (e.g. "[56, 78]")

Related Work

Since the time E.F. Codd published "A relational model of data for large shared data banks," [1] there has been a significant rise in databases adopting the relational data model methodology. The database marketplace has been dominated by organizations like IBM and Oracle that offer solutions based on relational databases. However, with time and rise of the World Wide Web, the type of data we now generate, store and use has changed. Relational database management systems, though successful, fail to capture these changes in data. Its limitations are very well explained in a blog by Michael Stonebraker [2] where he explains how relational database management systems can be beaten by other alternate database management systems. These include benefits of a column store instead of a row store for data warehouses where number of reads are more than writes, compression is effective and business intelligence queries are more optimised. Applications that involve the majority of data in the text format did not use relational database management systems. But the biggest drawback of these systems is that it becomes difficult to handle data that cannot be represented in the form of tables. The growth of the internet gave led to the generation of large scale data. This data was not uniform. While one business followed one structure, another followed a different one. Capturing all this data into one table made the table very sparse. Due to these drawbacks, relational database management systems were soon modified into or replaced by other database management systems.

In 2004, Google came out with MapReduce [3], a programming model for processing large scale datasets. The main goal of MapReduce was to provide efficient parallelization and distribution of large datasets on commodity hardware. But what the authors realized was that most of their computations involved executing a map operation to each logical record and generating an intermediate <key, value> pair. Furthermore, they apply a reduce function on all such pairs having a common key to combine them. Combining these pairs on the basis of a common enabled them to optimize their computations. It is important to notice that even when developing MapReduce, Google dealt with a large amount of unstructured data in the form of documents, logs, URLs and geographic locations to name a few. Classic relational database management systems are not well suited to store structured as well as semi- structured data. Along with maintaining URLs and logs, Google wanted to maintain versions of such data. Relational models do not offer any direct functionalities to store and compute on different versions of the same record. Motivated by these limitations, Google developed Bigtable[4], the focus of our project. Bigtable, is a distributed storage system, that provides flexibility and scalability for large scale data. Bigtable does not support a full relational model as it provides users with flexibility in storage and querying format. Our project however, is relatively closer to a relational model as it is built on top of Minibase. BigTable is a sparse system with maps as its basic element. A map is

composed of a row-label, column-label, timestamp and value. It can be indexed by a row key, column key and a timestamp. It also provides an API for creating and deleting tables as well as other functionalities to the user. It can be used with MapReduce to enable large scale data parallelization and distribution.

To further understand these different database management systems more clearly, we also studied one comparative work. The work [6] describes how atomicity, consistency, isolation, and durability (ACID) are a set of governing principles of the relational model as they guarantee persistence and reliability. NoSQL systems like bigtable on the other hand consider ACID properties to be highly expensive and reject them. Rather, they support the CAP theorem that states that out among consistency, availability and partition-tolerance only two can be maintained. We realised that while bigtable has many advantages, a relational data model also has certain functions that it excels at.

Yang *et al.* [5] highlighted that while systems like bigtable performed very well on homogenous data, they do not support direct functions like joins for heterogeneous datasets. Joining heterogeneous datasets using MapReduce is possible, but it leads to additional overheads. The authors suggest that one crucial improvement that systems like bigtable should undergo is providing relational algebra functionalities. The authors implement this using an approach called as Map-Reduce-Merger that offers a merger module that combines data that is already partitioned and sorted even in heterogeneous data. It does this by combining two keys from two different sets into a list of key, value output. This output then becomes a distinct entity of its own. The merge function takes a partitioned and then sorted(or hashed) dataset organised by key as input and then combines them like a two-dimensional list comprehension from functional programming. This approach offers multiple advantages. It retains the flexibility and simplicity of systems like bigtable but also offers additional relational operations on heterogeneous datasets. Furthermore this also enables users to reuse pre-existing relational algebra techniques.

After bigtable, Google came out with Megastore [7], another storage system that could meet the requirements of online services. It combines the scalability of a NoSQLsystem with the convenience of a relational database management system. This is because while relational databases can help easily build applications, systems like bigtable are highly flexible and scalable for large datasets. Hence, Megastore proposes a blend of both these database management systems. They adopt the middle ground between these two by partitioning the data store and replicating each one separately, hence guranteening ACID properties within them and consistency across them. It uses Paxos, a fault-tolerant algorithm, for replication. The data is partitioned into multiple datacenters. In each of these, the storage system is a bigtable. The data model is defined as a Megastore, which lies halfway between RDBMS and NoSQL. It is declared in a schema with tables that have entities with properties. But unlike a traditional

RDBMS, Megastore primary keys are made to cluster entities that will be read together. Each entity is mapped to a Bigtable row. The authors state that Megastore is capable of handling a large number of transactions that come up to more than 3 billion writes and 20 billion reads daily. They also state that Megastore has over 100 applications in production with both internal and external users. It stores nearly a petabyte of primary data across many global data centers. Megastore helps showcase how powerful a system developed by leveraging and adopting the advantages of a relational database system and a NoSQL system can be.

Conclusion

Bigtable[4] is a distributed storage system for managing structured data that is designed to scale to a very large size. This implementation uses the conventional RDBMS minibase as the base to build a bigtable-like DBMS to incorporate scalability, flexibility and sparseness properties of bigtable into the RDBMS style implementation.
RDBMS stores data in the form of fixed length tuples. Our implementation introduces map - <key,value> structure storage thereby adding flexibility and sparseness properties. However, it utilizes the existing tuple structure for directory pages.
Minibase is modified to count the number of disk reads and writes which helps to quantify the performance evaluation as described earlier.
Following factors would justify the performance enhancement:

- This implementation uses a clustered database structure giving an advantage for range queries over unclustered structure.
- It uses the external disk algorithm for sorting the maps, thereby making the implementation memory efficient.
- Disk accesses – Introduction of indexing significantly reduces the number of disk accesses as is evident by the result table.
  Insertion – The disk accesses reduce by 4 times when indexing is used.
  Query - For a non-indexed database (type 1), the disk accesses amount to approx. 3k for a range search or an equality search. These significantly reduce to approx. 800 when an indexed database is queried, for a range search with key same as the indexing and reduce to approx. 200, for an equality search.

While implementing and analysing joins for a bigtable-like DBMS we realise that system performance depends on the number of rows joined. When the number of joined rows is less, the system performs reasonably well. However, these operations are expensive and not well suited for a system where the number of joined rows is large.

Bibliography

1. Codd, Edgar F. "A relational model of data for large shared data banks." In Software pioneers, pp. 263-294. Springer, Berlin, Heidelberg, 2002.
2. Michael Stonebraker. “The End of a DBMS Era (Might Be Upon Us).” Communications of the ACM (2009).
3. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: a flexible data processing tool." Communications of the ACM 53, no. 1 (2010): 72-77.
4. Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26, no. 2 (2008): 1-26.
5. Yang, Hung-chih, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. "Map-reduce-merge: simplified relational data processing on large clusters." In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 1029-1040. 2007.
6. Vicknair, Chad, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. "A comparison of a graph database and a relational database: a data provenance perspective." In Proceedings of the 48th annual Southeast regional conference, pp. 1-6. 2010.
7. Baker, Jason, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing scalable, highly available storage for interactive services." (2011).

Appendix

Specific Roles of the group members

1. Abhilasha Mandal

Phase 2:

Iterator - MapUtils.java, OBuf.java

Phase 3:

Worked with Yash Sarvaiya to come up with an algorithm for rowsort.

2. Krisha Mehta

Phase 2:

BigT - HFPage.java

Overloaded the functionalities of tuples for maps as HFPage can be a directory page as well as a data page.

Diskmgr - bigDB.java, pcounter.java

Composite Keys

Index - IndexUtils.java

Read <String,String> composite key value from byte array

Global - Convert.java

Test.csv - Debugged the first three utf-8 encoding bytes that led to errors during testing.

Fixed a minibase error caused due to incorrectly initialized file directory pages. The new file directory pages weren't correctly initialized. This caused an error when sorting large files since they create numerous temporary bigT(s) which were added as file entries in the file directory pages. This was fixed by only attempting to read a file name if it did not belong to an invalid page.

Phase 3:

Implemented RowSort

Did a literature survey that included reading and studying papers to understand how to implement join functionality in bigtable like dbms.

3. Sai Madhuri Molleti

Phase 2:

Global - Mid.java

BatchInsert.java

Phase 3:

Tested BatchInsert

Tested MapInsert

Designed test queries to check Query functionality

4. Sanika Shinde

Phase 2:

Query Optimization

BigT/Stream.java

BatchInsert.java - Implemented, tested and fixed related bugs.

Query.java - Implemented, tested and fixed related bugs.

Designed test queries to exhaustively all index, stream, sort and scan components and analyse optimisation. (Collaborated with Yash Kotadia and Yash Sarvaiya)

Phase 3:

Implemented MapInsert.java

Functionality testing of MapInsert.

Modified following files to make deleteMap functionality work

Stream.java

BigStream.java

Studied the impact of current design changes on query optimizations implemented in the previous phase in Stream.java (collaborated with Yash Kotadia).

5. Yash Kotadia

Phase 2:

Wrote the definition of Map

BigT - Map

Global - GlobalConst

Query Optimization

BigT/Stream.java(collaborated with Sanika)

Composite Keys

Global - AttrType, AttrOperator

Btree - BT.java, BTreeFile.java, StringStringKey.java, KeyDataEntry.java

Persistence

Global - SysDefs

Insertion/Deletions in Index

BigT - bigT

Map Versioning (collaborated with Yash Sarvaiya)

BigT - bigT

Designed test queries to exhaustively all index, stream ,sort and scan components and analyse optimisation. (Collaborated with Yash Sarvaiya and Sanika)

Fixed a minibase inefficiency in BigT/bigT.java, which is a heap file based implementation of heap/HeapFile.java. In the getMap(mid) method Minibase originally scanned through the directory pages to find the datapage containing the required record. However since we already have MID which is a combination of page number and slot number, we can directly pin this page number and retrieve the map from the slot number contained in MID itself. This significantly increased the efficiency of IndexScan which uses getMap(mid) very frequently.

Phase 3:

Changed definition of BigTable and Disk Manager to support multiple storage and indexing schemes

BigT - bigT.java

Diskmgr - bigDB.java

Wrote the wrapper for combining streams from multiple heap files

BigT - BigStream.java

BatchInsert Operation

getCounts Query

Designed queries and various operations for Results and Analysis(collaborated with Yash Sarvaiya).

6. Yash Sarvaiya

Phase 2:

Map storage and manipulation:

BigT/bigT.java - map insertion, map deletion, map manipulation, getting distinct row and column labels, map versioning (collaborated with Yash Kotadia)

BigT/Scan.java - implementing sequential scan over bigT

BigT/DataPageInfo.java - altering the type of entries in the directory pages

Sorting: Disk based external sorting for maps

Iterator- Sort.java, pnode.java, pnodePQ.java, pnodeSplayPQ.java, SpoofIBuf.java, FileScan.java

ScanAndSortTest.java (Collaborated with Yash Kotadia) - Exhaustively testing sorting and and scanning.

Designed test queries to exhaustively test all index, stream ,sort and scan components and analyse optimisation. (Collaborated with Yash Kotadia and Sanika)

Fixed a minibase error which was not allowing sort to be completed on a large number of maps (when more temp files are created). It was not able to free all pages as some pages

had a pin count greater than one. It was fixed by creating a close method for the spoof input buffer (SpoofIBuf.java) for these temp files which closes its scan and is called in sort.close(). The SpoofIBuf was creating a scan and not closing it due to which all pages were not getting unpinned.

Phase 3:

Implemented rowjoin operator and wrote logic to perform a sort-merge join between two bigTables.

Classes created:

Rowjoin.java, BigSortMerge.java

Designed queries and various operations for Results and Analysis. (Collaborated with Yash Kotadia)