CSE 510 Project Phase 3

Sanchit Aggarwal
saggar26@asu.edu

Zhenyu Bao
zbao6@asu.edu

Matthew Behrendt
mbehren4@asu.edu

Ziming Dong
zdong27@asu.edu

Alexandra Szilagy
aszilagy@asu.edu

Hang Zhao
hzhao64@asu.edu

ABSTRACT

The purpose of this project phase is to add new functionalities to the first iteration of the Bigtable-like DBMS product. The new functionalities, the insertion of a single map, the ability to join the rows of two different databases, and the ability to sort a database based on its rows, allow the user to process their data in more ways than the first iteration. The second iteration improves the processing time of the first iteration, as well as introduces bug fixes that were present in the first iteration.

KEYWORDS

Bigtable, Maps, StorageType, RowJoin, RowSort

CSE 510 Project Phase 3

1 INTRODUCTION

The third and final phase of this project required us to modify some of the classes and structures upon which our Bigtable-like DBMS from the previous phase was built in addition to introducing new functionality to the system. Much of the core classes remained largely unchanged, but some modifications were required to make our Bigtable DBMS structure more flexible in how it is able to store data. In addition to the modification of these structural classes, further modifications were required for some of the functions implemented in the previous phase. Finally, new functions for working with the data in a more row-centric capacity were to be implemented as well.

1.1 Terminology

- Bigtable: Distributed DBMS that differs from traditional relational DBMSs by storing data using maps rather than tuples/rows.
- Rows: For rows in the BigTable, each row has maximum three maps, these three maps have same row and column name but different values and timestamps. Whenever a fourth map is added, it will replace the oldest map that has the same row and column name.
- Maps: Data structures represent records in a Bigtable-like DBMS. They consist of a row label, a column label, and a timestamp that are mapped to a value.
- Index Files: Structured files in a DBMS contain pointers to records in a table. It is sorted based on a specific attribute that has been denoted as the key.
- Data Files: Files in a DBMS containing the actual data for the records in a table. These were previously implemented as unsorted heap files in the prior phase, but have been modified in this phase to be sorted based on the index type.

1.2 Goal Description

As discussed briefly above, the overall goal for this project phase was to make changes to the structure of our Bigtable DBMS and some of its existing functionality, while also developing functions for implementing some new operations. The required structural modifications primarily entailed changing the bigDB and bigT classes so that they could accommodate multiple storage types in a single structure. Additionally, the definition of the five different storage types used in the previous phase underwent some changes requiring each type to have its data sorted based on the index key of its type. Following those modifications had been made, the function for the batchInsert function needed to be changed slightly, and a new function, mapInsert, was created for inserting a single map into a table. Similarly, the query program developed in the previous phase required slight modifications to accommodate the new structure and a slightly different set of input parameters, dropping the index type.

The project also required the implementation of a new function called "getCounts" that can be called from the command-line to return the number of maps, distinct row labels, and distinct column labels. The main new functionality that was to be implemented in this phase consisted of a RowJoin function and a RowSort function. The RowJoin operation required the creation of a new table containing the maps from rows that were joined based on the most recent value for the row labels on a given column value. RowSort sorts a given stream of maps in a similarly row-based fashion by ordering the sets of maps contained within each row based on that row label's most recent value for a given column label. All of the modifications and implementations required to accomplish the above tasks are described in detail in section 2.

1.3 Assumptions

Our implementation for this phase assumes that the data it is tested on is restricted to have row labels that can be combined to form strings of no more than 20 characters in length when two rows are joined. Additionally, we have assumed that there are similar restrictions on the lengths of the column labels to accommodate our appending of values to the column label strings denoting whether the map has come from the left or

right table for the implementation of the RowJoin operation. Aside from these, no other significant assumptions have been made for our implementation.

2 IMPLEMENTATION DETAILS

2.1 getCounts Implementation

Implementing the getCounts program was fairly straightforward, as the code for maintaining these count values had already been implemented in the previous phase. The values for the number of maps, the number of distinct row labels, and the number of distinct column labels are stored in variables in the bigT class which are updated as new maps are inserted. We created a function that prints out each of these values, and when the getCounts program is invoked from the command-line, we iterate through the existing tables and call this function for each of them.

2.2 Modification of bigDB and bigT Classes

The next step in our implementation of this project was to modify the bigDB and bigT classes so that the storage type was no longer an input parameter for the creation of an instance of either class, allowing them to contain sets of maps stored using a variety of different storage types. With the "type" parameter removed from the constructor, generalized, 'typeless' bigT instances could be created or opened for insertion, and batches of data or individual maps conforming to any possible storage type could be inserted into these tables alongside data from other types. Due to this change, we also revised our approach to searching for duplicate maps within a bigtable to ensure that, regardless of the type with which a certain batch of maps was inserted, only the three newest maps with the same row and column labels would be maintained throughout the entire table.

2.3 Modification of Storage Type

In addition to changing the primary structural classes to accommodate multiple storage types, the storage types themselves had to be modified as well. Storage type 1 remained unchanged, but types 2 through 5 had some additional restrictions on how the data was to be stored. Though the index corresponding to each of these types remained essentially the same aside from the removal of the timestamp indices from types 4 and 5, the data files themselves were now required to be sorted based on the map attribute on which the given index was built.

The implementation for this task entailed the creation of a sorting function using the index that would be called after insertion. Once the maps of an indexed type were inserted to a table, the index could then be used to retrieve the maps in sorted order based on the index key, allowing them to be re-inserted in the correct order. We were able to reduce the execution time significantly by skipping the process of searching for maps with the same row and column labels on re-insertion. This step could be skipped without failure to enforce the restriction on only having three maps with the same row and column labels since the old duplicates had already been removed accordingly in the initial insertion.

2.4 Modification of batchInsert

The batchInsert program from the previous phase also required some slight changes. Most of these changes, such as ensuring that maps with different storage types could be inserted into the same table and only the three newest maps with identical row and column labels were kept in a single table, were handled by the modifications to the bigT class discussed above in section 2.2. While most of the function was able to remain largely unchanged, we were required to accommodate a couple of changes. The first was a new input parameter, *NUMBUF*, that was introduced for this phase. Rather than using the global constant value of 50 from the previous phase, the batchInsert function instead reads this parameter from the command line and passes it in for the initialization of SystemDefs. Additionally, we removed the creation and insertion of timestamp indices for the insertion of storage types 4 and 5 since these indices were no longer part of the type definitions.

2.5 mapInsert Implementation

This phase also introduced a new program called "mapInsert," which would be used for inserting a single map into a table. Rather than receiving the path to a data file as input, the four fields of a map are provided instead. Aside from that, the other input parameters are the same as those for batchInsert and, as a result, most of the implementation for this function is essentially identical. We create a new byte array to serve as a pointer to the map, then use the setStrValue function to set

the fields of this array according to the input parameters representing the map's attributes. With this out of the way, the map is inserted into the table and its corresponding index following the same procedure as above.

2.6 Query Implementation

The query function gives the user the ability to search for maps that have specific values, so that they can gather meaningful data faster. They can input their desired database, order type, row filter, column filter, value filter, and number of buffer pages. The function processes the data by:

1) Retrieving the specified database.
2) Retrieving maps based on the index type that was given. For example, if the index type inputted is type 2, then the function retrieves maps from the b-tree that has an index that matches the row filter.
3) Compares the remaining filters to ensure the map satisfies every filter and inserts the map into an array to create a stream of maps.
4) Returns the stream of maps after all maps have been compared.
5) Prints the maps and the disk page read and disk page write count.

Each map has its filters compared using the function filterComp. The filterComp function has an input of label type, filter, and map. The label type parameter tells the function whether the filter is a row filter, column filter, or value filter. The filter parameter is can be a star, a range, or a single value. And finally the map is what the filter will be compared with. The function returns true if the maps label is in the range of, or equals the filter.

2.7 RowJoin Implementation

One of the major programs required in this phase implementation was RowJoin. Given the names of two tables to be joined, the name of an output table to store the results, a column name, and the number of buffer pages, this program performs a row-based join of all the maps belonging to row labels with the most recent maps, and the given column name has the same value. The RowJoin function works by:

1) Getting the map objects with the desired column label from both of the tables to be joined and storing them in separate lists.
2) Going through each of these lists and identifying the most recent values for all of the rows by keeping track of the row labels that have already been read and maintaining a list of arrays containing the row label, value, and timestamp of the most recent map for each row.
3) Iterating through these lists and comparing the most recent values for every row in the left table with those in the right table.
4) When a match has been found, a new row label is created by concatenating the two row labels with a colon in between them and the maps for both of these rows are inserted into a new table with this new row label and some additional possible modifications.
5) Insert map (explained in depth below)

The RowJoin function essentially accounts for three cases when inserting maps. When the map to be inserted has the column label on which the two tables are to be joined, the function inserts the map as normal, but it must check that no more than three maps with the row label and column label are in the table, keeping only the three most recent when this value exceeds three. The second case to account for is the insertion of maps with column labels that are shared by both of the rows to be joined. When this occurs, the function differentiates between the two rows by setting the column label as "ColumnName_L" when the map comes from the row in the left table and "ColumnName_R" when it comes from the right table. Finally, when the map has a column label that is unique to the row, the only necessary modification is to its row label, and it is inserted into the table otherwise unchanged.

2.8 RowSort Implementation

The RowSort will be used for rowsort function, which can sort the BigTable and use at most NUMBUF buffer pages to run the query. To begin with, we use the advantage of hashmap and and stack algorithm to implement the sorter. Since we only need to consider the increasing order, the first step will be to query the column name rows in the big table, for example:
Input Big Table has three rows:
California, Fox, 100, 1

California, Fox, 200, 2
California, Camel, 130, 4

Singapore, Fox, 100, 6
Singapore, Fox, 150, 7
Singapore, Camel, 300, 8

Greece, Camel, 200, 9
Greece, Giraffe, 300, 10
We split rows by the column name and only focus on the filter part, assuming the sort column name is "Fox", we get two rows:
California, Fox, 100, 1
California, Fox, 200, 2
Singapore, Fox, 100, 6
Singapore, Fox, 150, 7
Then the hashmap handles these rows, They are stored in an Arraylist at first, then create a map: <California,stack[records]>, the RowLabel name is the key, and all records from this row will be values, the values are stored in stack. Then it gets two maps with the keys: California,Singapore. The last step is to get the last value in each stack and store them in a list, in this example, the list will contain[200,150], because the most recent value for the California map is 200, and the most recent value for Singapore is 150. Then it sorts this list in increasing order, for each element in the list. Once the most recent value for this map is equal to this element, the whole map is queried. Finally, we can get the result:
Singapore, Fox, 100, 6
Singapore, Fox, 150, 7
California, Fox, 100, 1
California, Fox, 200, 2
After sorting the column name rows, we add the other rows to the top, then we get:
California, Camel, 130, 4
Singapore, Camel, 300, 8
Greece, Camel, 200, 9
Greece, Giraffe, 300, 10
California, Fox, 100, 1
California, Fox, 200, 2
Singapore, Fox, 100, 6
Singapore, Fox, 150, 7
At this moment, RowSorter work perfectly to sort the big table. Then we can use this funtion in the rowsort program by inputing INBTNAME, OUTBTNAME, column name and number of buffer pages to sort the table. The sorted table will be stored in the OUTBTNAME, when we query this table the sorted rows will be printed.

3 INTERFACE SPECIFICATIONS

Return Type	Signature
void	bigT(String name)
void	clearBuffer()
void	closeBigt()
void	deleteBigt()
void	deleteIndex(KeyClass key, MID mid, String name)
void	deleteRecord(MID mid, int type)
void	fileSort (int fileType)
void	getCount()
MAP	getMap(MID mid, int type)
void	insertIndex(KeyClass key, MID mid, String name)
MID	insertMap(String name, int type, byte[] mapPtr)
Stream	openStream(bigT db, String name, int ordertype, String rowFilter, String columnFilter, String valueFilter, int numbuf)
ArrayList<MID>	scanForKey(String name, KeyClass Key, KeyClass Key, int type)
Map<MID,MAP>	search(byte[] mapPtr)

Table 1: bigT Methods

Return Type	Signature
void	Stream(bigT db, String name, int ordertype, String rowFilter, String columnFilter, String valueFilter, int NUMBUF)
ArrayList<MAP>	query(bigT db, String rowFilter, String columnFilter, String valueFilter)
boolean	filterComp(int labelType, String filter, MAP bmap)
void	closestream()
void	reset()
void	getNext()
ArrayList<MAP>	getArray()

Table 2: Stream Methods

Return Type	Signature
int	CompareMapWithMap(MAP m1, MAP m2, int map_fld_no)
boolean	Equal(MAP m1, MAP m2)
boolean	compareRowCol(MAP m1, MAP m2)
String	Value(MAP map, int fldno)
void	SetValue(MAP value, MAP map, int fld_no)

Table 3: MapUtils Methods

Return Type	Signature
void	MAP()
void	MAP(int size)
void	MAP(byte[] amap)
void	MAP(MAP fromMap)
String	getRowLabel()
String	getColumnLabel()
int	getTimeStamp()
String	getValue()
void	setRowLabel(String val)
void	setColumnLabel(String val)
void	setTimeStamp(int val)
void	setValue(String val)
byte[]	getMapByteArray()
byte[]	getMapData()
void	print()
int	size()
void	mapCopy(MAP fromMap)
void	mapInit(byte[] amap)
void	mapSet(byte[] frommap, int offset)
MAP	setIntFld(int fldNo, int val)
MAP	setFloFld(int fldNo, float val)
MAP	setStrFld(int fldNo, String val)

Table 4: MAP Methods

Return Type	Signature
void	initialize()
void	readInc()
void	writeInc()
int	getreadCount()
int	getwriteCount()

Table 5: pcounter Methods

4 INSTALLATION AND EXECUTION INSTRUCTIONS

The instructions to run the minibase and test its functionalities are as follows-

1) Copy the “CSE510.tar.gz” file to your account in a Unix/Linux based system.
2) Uncompress the file using the command- "tar -xvf CSE510.tar"
3) Modify the ’Makefiles’ to reflect your directory structure.
4) In the "src" directory, build the source by running the command - "make db"
5) Now to start the program from the "src" directory, use the command - "make menu"
6) This previous command will print a list of options.

Execute any command listed in the options in the given format to achieve desired results. To exit the program, simply press Ctrl+C or type ’exit’ in the command line. The command specifications showed in the menu are as follows-

1. Batch Insert:
batchinsert [DataFile] [IndexType] [DB_Name] [NumBuff]

The batch insert function can be invoked from the command line using the above format. The parameters like ’DataFile’ defined the location of the csv data file from which the data has to be extracted and inserted into the database. ’IndexType’ defines the type of index to be generated for that table. There can be 5 types of indexes-

- Type 1: No index
- Type 2: Index and sort on row labels
- Type 3: Index and sort on column labels
- Type 4: Index and sort on column label and row label (combined key)
- Type 5: Index and sort on row label and value (combined key) and one index on timestamps

Use the integer value corresponding to the type needed. The ’DB_Name’ is the Big Table Name in which the table has to be inserted. And the ’NumBuff’ placeholder is to define the number of buffer pages to be used.

2. Query:
query [DB_Name] [OrderType] [RowFilter] [ColumnFilter] [ValueFilter] [NumBuff]

The parameters in query includes search key filters like, ’RowFilter’, ’ColumnFilter’, ’ValueFilter’ to find maps with equality search or range search based on the values given for the search keys. To skip using any search key, use- ’*’. To specify a range from x to y, use the format- ’[x,y]’. The ’DB_Name’ defines the Big Table name in which the maps have to be searched in. The ’NumBuff’ parameter is used to set the number of buffer pages to be used for the query. There will be no index type needed in the command in this phase. The ’OrderType’ defines the order in which the results have to be sorted. The order type take the integer value of the corresponding types mentioned below-

- Type 1: Results are first ordered in row label, then column label, then time stamp.
- Type 2: Results are first ordered in column label, then row label, then time stamp.
- Type 3: Results are first ordered in row label, then time stamp.
- Type 4: Results are first ordered in column label, then time stamp.
- Type 5: Results are ordered in time stamp.

3. Map Insert:
mapinsert [RowLabel] [ColumnLabel] [Value] [TS] [Type] [BTName] [Numbuff]

The parameters used in this command are 'RowLabel', 'ColumnLabel', 'Value', 'TS' which are the values to be inserted and 'Type' is the index type to which partition this record will be added, 'BTName' is the table name to which the map is being inserted and 'Numbuff' is the number of buffer pages to be used for the operation.

4. Get counts:
getcount

This command will return the numbers of maps, distinct row labels, and distinct column labels.

5. Row Join:
rowjoin [BT1] [BT2] [OUTBTNAME] [ColumnFilter] [Numbuff]

The parameters used in this command are 'BT1'and 'BT2', which are the two tables from which rows will be joined. 'OUTBTNAME' is the table name to which the results will be added with type 1 index format. 'ColumnFilter' is the column label on which 'BT1' and 'BT2' are to be joined. 'Numbuff' is the number of buffer pages to be used for the operation.

6. Row Sort:
rowsort [INBTNAME] [OUTBTNAME] [ColumnLabel] [NumBuff]

The parameters used in this command are 'INBTNAME', which is the database which we used for batch insert, 'OUTBTNAME' is the name of the output table that will be created to save the results. 'ColumnLabel' is the column label on which the dataset will be sorted according to it's most recent values. The 'NumBuff' parameter is the number of buffer pages that can be used to run this command. To check the results, query the 'OUTBTNAME' with ordertype 0, and the sorted maps can be observed there.

5 RELATED WORK

With Bigtable growing extremely quickly in popularity as a flexible alternative to traditional DBMSs over the past decade, a lot of research into how to most effectively leverage this system has been conducted. A 2013 paper by Haiping Wang, Xiang Ci, and Xiaofeng Meng explores a potential way to optimize the performance of Bigtable systems for performing queries on multiple fields [1]. In this paper, they propose a new method of allocating data and processing queries in a way that takes better advantage of the system's properties to make this process more effective.

Essentially, their proposed data allocation approach consists of using a set of rules for selecting certain fields to use for generating row keys and organizing the regions of the table using a multi-layer grid tree structure with the intent to keep the number of splits required and the resulting depth of the tree at a minimum. The query process itself is optimized through the use of query decomposition based on the fields related to the query and its constraints in conjunction with the new strategy for data allocation to identify candidate regions in the tree structure, formulate a query plan, and use MapReduce to execute this plan. Query performance is further improved through modifications to the existing approach to region allocation and the use of multi-threading to execute portions of the query in parallel.

At the same time, other scalable, dynamic databases have been developed as well. A paper by Ravinsingh Jain, Srikant Iyengar, and Ananyaa Arora provides a brief survey of prominent graph-based database systems that depart from the traditional relational model [2]. One of these other systems is Cassandra, a very similar map-based system developed by Facebook. The underlying structure of Cassandra's approach is made up of row keys and column families, much like Bigtable, with one of the core differences being the distinction

between two types of column families, which can be super column families or simple column families. The former essentially represents column families that are contained within other families, whereas the latter represents standard column families. They also discuss Amazon's DynamoDB as another non-traditional DBMS. This system uses a noSQL-based key-value approach to storing data and offers a configurable backend that allows for an administrator-specified number of reads and writes to the DB each second, making it easily scalable.

6 CONCLUSIONS AND EXPLANATION OF TEST RESULTS

Batch insert was ran for all index types with both 50 buffer frames and 100 buffer frames. The first set is inserted into dbtest1 and the second set is inserted into dbtest2. By looking at the results we can see that the run time is significantly longer for index type 2 to 5 due to sorting. Sequential batches also take more time because more maps need to be searched to remove duplicates. Increasing the buffer frame size from 50 to 100 also made a difference in time as we can see the insertion time for index type 1 decreased from 8704 milliseconds to 7253 milliseconds after doubling the buffer frame size. The result was similar for single map inserts. Although insertion time is about the same between different index types and number of buffer frames, run time is significantly longer for index type 2 to 5 due to resorting of the table after each insert query.

Query execution is similar to the previous phase and it can filter the maps by row, column, value and time stamp filters. There are a few page writes observed in this phase because data is partitioned with different index types and sort order within the table and the intermediate results needs to be sorted which requires temporary pages for this operation. The results are printed along the number of maps, disk page reads and writes and the time taken by the operation. In fact, to compare these results when we used different filter types and buffer sizes, we can find those cost did not change by a significant amount.

In this phase, we implement two new operations-RowJoin and RowSort. For RowJoin, we utilize buffer frames, the names of two existing databases to be joined, and the name of the new database being created by this function as well as a column filter. The performance of this operation was quite slow with large test data-sets. The first join that we tested, which used 50 buffer frames and resulted in 7831 maps in the new joined table, took 211214 milliseconds, or just over three and a half minutes. Subsequent tests of the join operation yielded similar results. This is due to our use of an approach based on nested loop join which becomes very inefficient with large data-sets. Future iterations of this project would definitely be well-served to use a more robust method such as sort-merge or hash join, which would likely reduce this execution time considerably.

The row sort operation sorted the maps in one database by column filter, then saves the results in a new big table. If we use the same database as target but a different buffer size to sort, the 50 buffer frames will take 8022 ms and the 100 buffer frames will take 8410 ms. Otherwise, the 50 frames need write 17910 disk pages, but the 100 frames only need 17116 pages. Hence, the 50 buffer frame will use more disk pages to write, but less time to execute command.

REFERENCES

[1] H. Wang, X. Ci, and X. Meng, "Fast multi-fields query processing in bigtable based cloud systems," in *Web-Age Information Management*, J. Wang, H. Xiong, Y. Ishikawa, J. Xu, and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 142–154.

[2] R. Jain, S. Iyengar, and A. Arora, "Overview of popular graph databases," in *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 2013, pp. 1–6.

A SPECIFICATION FOR ROLES OF THE GROUP MEMBERS

- Sanchit Aggarwal: Batch Insert, Command line invocation program, Integration, Testing, Report
- Zhenyu Bao: Testing, Report
- Matthew Behrendt: RowJoin, testing, report
- Ziming Dong: RowSorter operation and rowsort program, testing and fix bugs, writing report.
- Alexandra Szilagy: Query, pcounter, RowJoin, report
- Hang Zhao: File structure, BigDB, BigT, BatchInsert, MapInsert, GetCount, Testing and collecting results for report.

B TEST RESULTS

B.1 Batch Insert

Buffer Frames: 50
DB: dbtest1
Index: Type 1
Dataset: Data1.csv
batchinsert ../Data1.csv 1 dbtest1 50
DB: dbtest1
Map count: 9777
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 1617119
Number of disk page writes: 24644
This process took: 8704 milliseconds

Index: Type 2
Dataset: Data2.csv
batchinsert ../Data2.csv 2 dbtest1 50
DB: dbtest1
Map count: 17726
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 16498903
Number of disk page writes: 164436
This process took: 49222 milliseconds

Index: Type 3
Dataset: Data3.csv
batchinsert ../Data3.csv 3 dbtest1 50
DB: dbtest1
Map count: 23038
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 19387760
Number of disk page writes: 190545
This process took: 55188 milliseconds

Index: Type 4
Dataset: Data1.csv
batchinsert ../Data1.csv 4 dbtest1 50
DB: dbtest1
Map count: 24768
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 21737750
Number of disk page writes: 266631
This process took: 59418 milliseconds

Index: Type 5
Dataset: Data2.csv
batchinsert ../Data2.csv 5 dbtest1 50
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 21502970
Number of disk page writes: 288049
This process took: 58955 milliseconds

Buffer Frames: 100
DB: dbtest2
Index: Type 1
Dataset: Data1.csv
batchinsert ../Data1.csv 1 dbtest2 100
DB: dbtest2
Map count: 9777
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 1579354
Number of disk page writes: 23508
This process took: 7253 milliseconds

Index: Type 2
Dataset: Data2.csv
batchinsert ..Data2.csv 2 dbtest2 100
DB: dbtest2
Map count: 17726
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 16311967
Number of disk page writes: 159114
This process took: 45901 milliseconds

Index: Type 3
Dataset: Data3.csv
batchinsert ../Data3.csv 3 dbtest2 100
DB: dbtest2
Map count: 23038
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 19186171
Number of disk page writes: 185378
This process took: 52449 milliseconds

Index: Type 4

Dataset: Data1.csv
batchinsert ../Data1.csv 4 dbtest2 100
DB: dbtest2
Map count: 24768
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 21533713
Number of disk page writes: 258067
This process took: 61353 milliseconds

Index: Type 5
Dataset: Data5.csv
batchinsert ../Data2.csv 5 dbtest2 100
DB: dbtest2
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 21247863
Number of disk page writes: 279069
This process took: 59183 milliseconds

Buffer Frames: 25
DB: dbtest3
Index: Type 1
batchinsert ../Data1.csv 1 dbtest3 25
DB: dbtest3
Map count: 9777
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 1634745
Number of disk page writes: 25389
This process took: 7493 milliseconds

B.2 Map Insert

Buffer Frames: 50
DB: dbtest1
Index: Type 1
mapinsert Dominica Zebra 1 1 dbtest1 1 50
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 1537
Number of disk page writes: 16
This process took: 709 milliseconds

Index: Type 2
mapinsert Dominica Zebra 2 2 dbtest1 2 50
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 4249861
Number of disk page writes: 46010
This process took: 11987 milliseconds

Index: Type 3
mapinsert Dominica Zebra 3 3 dbtest1 3 50
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 5394234
Number of disk page writes: 58211
This process took: 14355 milliseconds

Index: Type 4
mapinsert Dominica Zebra 4 4 dbtest1 4 50
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 4214795
Number of disk page writes: 70200
This process took: 11938 milliseconds

Index: Type 5
mapinsert Dominica Zebra 5 5 dbtest1 5 50
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 6639332
Number of disk page writes: 99342
This process took: 18757 milliseconds

Buffer Frames: 100
DB: dbtest2
Index: Type 1
mapinsert Dominica Zebra 1 1 dbtest2 1 100
DB: dbtest2
Map count: 25994
Distinct row count: 100

Distinct column count: 99
Number of disk page reads: 1542
Number of disk page writes: 16
This process took: 793 milliseconds

Index: Type 2
mapinsert Dominica Zebra 2 2 dbtest2 2 100
DB: dbtest2
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 4208870
Number of disk page writes: 44447
This process took: 12765 milliseconds

Index: Type 3
mapinsert Dominica Zebra 3 3 dbtest2 3 100
DB: dbtest2
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 5327991
Number of disk page writes: 56076
This process took: 15097 milliseconds

Index: Type 4
mapinsert Dominica Zebra 4 4 dbtest2 4 100
DB: dbtest2
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 4129168
Number of disk page writes: 66030
This process took: 11929 milliseconds

Index: Type 5
mapinsert Dominica Zebra 5 5 dbtest2 5 100
DB: dbtest2
Map count: 25994
Distinct row count: 100
Distinct column count: 99
Number of disk page reads: 6497711
Number of disk page writes: 93546
This process took: 19860 milliseconds

B.3 Query

Buffer Frames: 50
DB: dbtest1
Order: Type 1
query dbtest1 1 [A,Ar] [E,G] * 50
DB name: dbtest1
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Fish, 27734, 27818]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Fox, 6726, 14448]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alaska, Echinorhi, 90666, 26113]
[Alaska, Fish, 59524, 13768]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 95164, 29825]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
Number of maps found: 21
Number of disk page reads: 2218088
Number of disk page writes: 432
This process took: 5708 milliseconds

Order: Type 2
query dbtest1 2 [A,Ar] [E,G] * 50
DB name: dbtest1
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alaska, Echinorhi, 90666, 26113]
[Alabama, Fish, 27734, 27818]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alaska, Fish, 59524, 13768]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]

[Alabama, Flamingo, 59588, 13623]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 95164, 29825]
[Alabama, Fox, 6726, 14448]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
Number of maps found: 21
Number of disk page reads: 2218088
Number of disk page writes: 432
This process took: 5580 milliseconds

Order: Type 3
query dbtest1 3 [A,Ar] [E,G] * 50
DB name: dbtest1
[Alabama, Fox, 6726, 14448]
[Alabama, Fish, 27734, 27818]
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alaska, Fish, 59524, 13768]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Echinorhi, 90666, 26113]
[Alaska, Flamingo, 95164, 29825]
Number of maps found: 21
Number of disk page reads: 2218088
Number of disk page writes: 432
This process took: 5930 milliseconds

Order: Type 4
query dbtest1 4 [A,Ar] [E,G] * 50
DB name: dbtest1
[Alabama, Echinorhi, 48246, 19014]
[Alaska, Echinorhi, 90666, 26113]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Fish, 27734, 27818]
[Alaska, Fish, 59524, 13768]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 95164, 29825]
[Alabama, Fox, 6726, 14448]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
Number of maps found: 21
Number of disk page reads: 2218088
Number of disk page writes: 432
This process took: 5452 milliseconds

Order: Type 5
query dbtest1 5 [A,Ar] [E,G] * 50
DB name: dbtest1
[Alabama, Fox, 6726, 14448]
[Alabama, Fish, 27734, 27818]
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Flamingo, 55673, 18430]
[Alaska, Fish, 59524, 13768]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Echinorhi, 90666, 26113]
[Alaska, Flamingo, 95164, 29825]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]

Number of maps found: 21
Number of disk page reads: 2218088
Number of disk page writes: 432
This process took: 6103 milliseconds

Buffer Frames: 100
DB: dbtest2
Order: Type 1
query dbtest2 1 [A,Ar] [E,G] * 100
DB name: dbtest2
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Fish, 27734, 27818]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Fox, 6726, 14448]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alaska, Echinorhi, 90666, 26113]
[Alaska, Fish, 59524, 13768]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 95164, 29825]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
Number of maps found: 21
Number of disk page reads: 2172643
Number of disk page writes: 433
This process took: 5918 milliseconds

Order: Type 2
query dbtest2 2 [A,Ar] [E,G] * 100
DB name: dbtest2
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alaska, Echinorhi, 90666, 26113]
[Alabama, Fish, 27734, 27818]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alaska, Fish, 59524, 13768]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 95164, 29825]
[Alabama, Fox, 6726, 14448]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
Number of maps found: 21
Number of disk page reads: 2172643
Number of disk page writes: 433
This process took: 5466 milliseconds

Order: Type 3
query dbtest2 3 [A,Ar] [E,G] * 100
DB name: dbtest2
[Alabama, Fox, 6726, 14448]
[Alabama, Fish, 27734, 27818]
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alaska, Fish, 59524, 13768]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Echinorhi, 90666, 26113]
[Alaska, Flamingo, 95164, 29825]
Number of maps found: 21
Number of disk page reads: 2172643
Number of disk page writes: 433
This process took: 5477 milliseconds

Order: Type 4
query dbtest2 4 [A,Ar] [E,G] * 100

DB name: dbtest2
[Alabama, Echinorhi, 48246, 19014]
[Alaska, Echinorhi, 90666, 26113]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Fish, 27734, 27818]
[Alaska, Fish, 59524, 13768]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alabama, Flamingo, 55673, 18430]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 95164, 29825]
[Alabama, Fox, 6726, 14448]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
Number of maps found: 21
Number of disk page reads: 2172643
Number of disk page writes: 433
This process took: 5630 milliseconds

Order: Type 5
query dbtest2 5 [A,Ar] [E,G] * 100
DB name: dbtest2
[Alabama, Fox, 6726, 14448]
[Alabama, Fish, 27734, 27818]
[Alabama, Echinorhi, 48246, 19014]
[Alabama, Flamingo, 55673, 18430]
[Alaska, Fish, 59524, 13768]
[Alabama, Flamingo, 59588, 13623]
[Alabama, Flamingo, 59588, 13623]
[Alaska, Fox, 61563, 13195]
[Alaska, Fox, 61563, 13195]
[Alabama, Fish, 62393, 5059]
[Alabama, Fish, 62393, 5059]
[Alaska, Fish, 65944, 2108]
[Alaska, Fish, 65944, 2108]
[Alabama, Fox, 71959, 1158]
[Alabama, Fox, 71959, 1158]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Flamingo, 83861, 12453]
[Alaska, Echinorhi, 90666, 26113]
[Alaska, Flamingo, 95164, 29825]
[Alabama, Echinorhi, 97664, 6960]
[Alabama, Echinorhi, 97664, 6960]
Number of maps found: 21
Number of disk page reads: 2172643
Number of disk page writes: 433
This process took: 5428 milliseconds

B.4 Row Sort

Buffer Frames: 50
Sort DB: dbtest3
Output DB: dbsort1
Column Filer: Zebra
rowsort dbtest3 dbsort1 Zebra 50
Number of disk page reads: 1868527
Number of disk page writes: 17910
This process took: 8022 milliseconds

query dbsort1 0 * Zebra * 50
DB name: dbsort1
[Ghana, Zebra, 36965, 20]
[Luxembour, Zebra, 3819, 362]
[Japan, Zebra, 12796, 422]
[Serbia, Zebra, 59184, 526]
[Maryland, Zebra, 6032, 2562]
[Maryland, Zebra, 55915, 636]
[EQUATORIA, Zebra, 48592, 701]
[Rhode$_{I}$*sl*, *Zebra*, 44819, 897]
[*Monaco*, *Zebra*, 11692, 7585]
[*Monaco*, *Zebra*, 91108, 1575]
[*Dominica*, *Zebra*, 86890, 1582]
[*Sweden*, *Zebra*, 36261, 6253]
[*Sweden*, *Zebra*, 68453, 1727]
[*Saudi*$_{A}$*ra*, *Zebra*, 18537, 1891]
[*Grenada*, *Zebra*, 71893, 1976]
[*Nigeria*, *Zebra*, 38002, 2168]
[*Malawi*, *Zebra*, 29223, 2366]
[*Alabama*, *Zebra*, 50568, 2437]
[*Virginia*, *Zebra*, 24594, 2658]
[*NEW_ZEALA*, *Zebra*, 14602, 7649]
[*NEW_ZEALA*, *Zebra*, 23866, 2721]
[*Louisiana*, *Zebra*, 4409, 1946]
[*Louisiana*, *Zebra*, 55080, 3053]
[*Ukraine*, *Zebra*, 29218, 3056]
[*New_Jerse*, *Zebra*, 58296, 3113]

[$Vatican_C$, $Zebra$, 72896, 3226]
[$Denmark$, $Zebra$, 32046, 3294]
[$Colombia$, $Zebra$, 63378, 7577]
[$Colombia$, $Zebra$, 90587, 3494]
[$Georgia$, $Zebra$, 84319, 4131]
[$CZECH_REP$, $Zebra$, 16368, 7612]
[$CZECH_REP$, $Zebra$, 60050, 5306]
[$CZECH_REP$, $Zebra$, 60590, 4139]
[$Texas$, $Zebra$, 9226, 4080]
[$Texas$, $Zebra$, 28496, 4307]
[$Tuvalu$, $Zebra$, 2517, 6799]
[$Tuvalu$, $Zebra$, 74902, 2482]
[$Tuvalu$, $Zebra$, 90204, 4406]
[$Mississip$, $Zebra$, 46618, 4482]
[$Singapore$, $Zebra$, 52432, 4634]
[$Ireland$, $Zebra$, 68628, 9418]
[$Ireland$, $Zebra$, 85673, 4680]
[$Namibia$, $Zebra$, 52332, 4700]
[$Guatemala$, $Zebra$, 64486, 4702]
[$Latvia$, $Zebra$, 25255, 4782]
[$Pennsylva$, $Zebra$, 11670, 4829]
[$Sudan$, $Zebra$, 63852, 4900]
[$Finland$, $Zebra$, 34584, 5295]
[$Minnesota$, $Zebra$, 68220, 5614]
[$COSTA\_RIC$, $Zebra$, 28771, 5634]
[$Ecuador$, $Zebra$, 68275, 5728]
[$Morocco$, $Zebra$, 49929, 6301]
[$Nepal$, $Zebra$, 21149, 6417]
[$Jordan$, $Zebra$, 42946, 6567]
[$Tonga$, $Zebra$, 9042, 9991]
[$Tonga$, $Zebra$, 49953, 2648]
[$Tonga$, $Zebra$, 60491, 6789]
[$Thailand$, $Zebra$, 63158, 6889]
[$Netherlan$, $Zebra$, 77870, 7108]
[$Madagasca$, $Zebra$, 41328, 7397]
[$France$, $Zebra$, 2250, 7683]
[$France$, $Zebra$, 47813, 7488]
[$Barbados$, $Zebra$, 44641, 7951]
[$Samoa$, $Zebra$, 189, 7961]
[$Nauru$, $Zebra$, 3059, 7988]
[$Oregon$, $Zebra$, 19347, 4268]
[$Oregon$, $Zebra$, 76772, 8277]
[$Illinois$, $Zebra$, 30098, 8694]
[$Slovakia$, $Zebra$, 84165, 8774]
[$Tunisia$, $Zebra$, 81730, 8873]
[$Arizona$, $Zebra$, 7389, 9941]
$Numberofmapsfound$: 71
$Numberofdiskpagereads$: 767
$Numberofdiskpagewrites$: 0
$Thisprocesstook$: 769$milliseconds$

$BufferFrames$: 100
$SortDB$: $dbtest3$
$OutputDB$: $dbsort2$
$ColumnFiler$: $Zebra$
$rowsortdbtest3dbsort2Zebra100$
$Numberofdiskpagereads$: 1853621
$Numberofdiskpagewrites$: 17116
$Thisprocesstook$: 8410$milliseconds$

$querydbsort20 * Zebra * 100$
$DBname$: $dbsort2$
[$Ghana$, $Zebra$, 36965, 20]
[$Luxembour$, $Zebra$, 3819, 362]
[$Japan$, $Zebra$, 12796, 422]
[$Serbia$, $Zebra$, 59184, 526]
[$Maryland$, $Zebra$, 6032, 2562]
[$Maryland$, $Zebra$, 55915, 636]
[$EQUATORIA$, $Zebra$, 48592, 701]
[$Rhode_Isl$, $Zebra$, 44819, 897]
[$Monaco$, $Zebra$, 11692, 7585]
[$Monaco$, $Zebra$, 91108, 1575]
[$Dominica$, $Zebra$, 86890, 1582]
[$Sweden$, $Zebra$, 36261, 6253]
[$Sweden$, $Zebra$, 68453, 1727]
[$Saudi\_Ara$, $Zebra$, 18537, 1891]
[$Grenada$, $Zebra$, 71893, 1976]
[$Nigeria$, $Zebra$, 38002, 2168]
[$Malawi$, $Zebra$, 29223, 2366]
[$Alabama$, $Zebra$, 50568, 2437]
[$Virginia$, $Zebra$, 24594, 2658]
[NEW_ZEALA, $Zebra$, 14602, 7649]
[NEW_ZEALA, $Zebra$, 23866, 2721]
[$Louisiana$, $Zebra$, 4409, 1946]
[$Louisiana$, $Zebra$, 55080, 3053]
[$Ukraine$, $Zebra$, 29218, 3056]
[New_Jerse, $Zebra$, 58296, 3113]
[$Vatican_C$, $Zebra$, 72896, 3226]
[$Denmark$, $Zebra$, 32046, 3294]
[$Colombia$, $Zebra$, 63378, 7577]
[$Colombia$, $Zebra$, 90587, 3494]
[$Georgia$, $Zebra$, 84319, 4131]
[$CZECH_REP$, $Zebra$, 16368, 7612]
[$CZECH_REP$, $Zebra$, 60050, 5306]
[$CZECH_REP$, $Zebra$, 60590, 4139]
[$Texas$, $Zebra$, 9226, 4080]

[*Texas*, *Zebra*, 28496, 4307]
[*Tuvalu*, *Zebra*, 2517, 6799]
[*Tuvalu*, *Zebra*, 74902, 2482]
[*Tuvalu*, *Zebra*, 90204, 4406]
[*Mississip*, *Zebra*, 46618, 4482]
[*Singapore*, *Zebra*, 52432, 4634]
[*Ireland*, *Zebra*, 68628, 9418]
[*Ireland*, *Zebra*, 85673, 4680]
[*Namibia*, *Zebra*, 52332, 4700]
[*Guatemala*, *Zebra*, 64486, 4702]
[*Latvia*, *Zebra*, 25255, 4782]
[*Pennsylva*, *Zebra*, 11670, 4829]
[*Sudan*, *Zebra*, 63852, 4900]
[*Finland*, *Zebra*, 34584, 5295]
[*Minnesota*, *Zebra*, 68220, 5614]
[*COSTA_RIC*, *Zebra*, 28771, 5634]
[*Ecuador*, *Zebra*, 68275, 5728]
[*Morocco*, *Zebra*, 49929, 6301]
[*Nepal*, *Zebra*, 21149, 6417]
[*Jordan*, *Zebra*, 42946, 6567]
[*Tonga*, *Zebra*, 9042, 9991]
[*Tonga*, *Zebra*, 49953, 2648]
[*Tonga*, *Zebra*, 60491, 6789]
[*Thailand*, *Zebra*, 63158, 6889]
[*Netherlan*, *Zebra*, 77870, 7108]
[*Madagasca*, *Zebra*, 41328, 7397]
[*France*, *Zebra*, 2250, 7683]
[*France*, *Zebra*, 47813, 7488]
[*Barbados*, *Zebra*, 44641, 7951]
[*Samoa*, *Zebra*, 189, 7961]
[*Nauru*, *Zebra*, 3059, 7988]
[*Oregon*, *Zebra*, 19347, 4268]
[*Oregon*, *Zebra*, 76772, 8277]
[*Illinois*, *Zebra*, 30098, 8694]
[*Slovakia*, *Zebra*, 84165, 8774]
[*Tunisia*, *Zebra*, 81730, 8873]
[*Arizona*, *Zebra*, 7389, 9941]
Numberofmapsfound : 71
Numberofdiskpagereads : 768
Numberofdiskpagewrites : 0
Thisprocesstook : 750*milliseconds*

B.5 Row Join

Buffer Frames: 50
Join DB: dbtest3 + dbtest1
Output DB: joindb
Column Filer: Zebra
rowjoin dbtest3 dbtest1 joindb Zebra 50
DB: joindb
Map count: 7831
Distinct row count: 22
Distinct column count: 295
Number of disk page reads: 97559380
Number of disk page writes: 55027
This process took: 211214 milliseconds

Buffer Frames: 100
Join DB: dbtest3 + dbtest1
Output DB: joindb
Column Filer: Zebra
rowjoin dbtest3 dbtest1 joindb Zebra 100
DB: joindb
Map count: 9389
Distinct row count: 22
Distinct column count: 295
Number of disk page reads: 102738897
Number of disk page writes: 73590
This process took: 229589 milliseconds

Buffer Frames: 50
Join DB: dbtest1 + dbtest3
Output DB: joindb
Column Filer: Zebra
rowjoin dbtest1 dbtest3 joindb Zebra 50
DB: joindb
Map count: 10275
Distinct row count: 22
Distinct column count: 295
Number of disk page reads: 110062208
Number of disk page writes: 78135
This process took: 244554 milliseconds

Buffer Frames: 100
Join DB: dbtest1 + dbtest3
Output DB: joindb
Column Filer: Zebra
rowjoin dbtest1 dbtest3 joindb Zebra 100
DB: joindb
Map count: 10275
Distinct row count: 22
Distinct column count: 295
Number of disk page reads: 108230863
Number of disk page writes: 77773

This process took: 248469 milliseconds

Buffer Frames: 50
DB: joindb
Order: Type 1
query jointable 1 * Zebra * 50
DB name: jointable
[Alabama:Alabama, Zebra, 50568, 2437]
[Alabama:Alabama, Zebra, 50568, 2437]
[Alabama:Alabama, Zebra, 50568, 2437]
[CZECH_REP:CZECH_REP, Zebra, 60590, 4139]
[CZECH_REP:CZECH_REP, Zebra, 60590, 4139]
[CZECH_REP:CZECH_REP, Zebra, 60590, 4139]
[Colombia:Colombia, Zebra, 63378, 7577]
[Colombia:Colombia, Zebra, 90587, 3494]
[Colombia:Colombia, Zebra, 90587, 3494]

[Dominica:Dominica, Zebra, 86890, 1582]
[Dominica:Dominica, Zebra, 86890, 1582]
[Dominica:Dominica, Zebra, 86890, 1582]
[Georgia:Georgia, Zebra, 84319, 4131]
[Georgia:Georgia, Zebra, 84319, 4131]
[Georgia:Georgia, Zebra, 84319, 4131]
[Grenada:Grenada, Zebra, 71893, 1976]
[Grenada:Grenada, Zebra, 71893, 1976]
[Grenada:Grenada, Zebra, 71893, 1976]
[Guatemala:Guatemala, Zebra, 64486, 4702]
[Guatemala:Guatemala, Zebra, 64486, 4702]
[Guatemala:Guatemala, Zebra, 64486, 4702]
[Ireland:Ireland, Zebra, 85673, 4680]
[Ireland:Ireland, Zebra, 85673, 4680]
[Ireland:Ireland, Zebra, 85673, 4680]
[Latvia:Latvia, Zebra, 25255, 4782]
[Latvia:Latvia, Zebra, 25255, 4782]
[Latvia:Latvia, Zebra, 25255, 4782]
[Madagasca:Madagasca, Zebra, 41328, 7397]
[Madagasca:Madagasca, Zebra, 41328, 7397]
[Madagasca:Madagasca, Zebra, 41328, 7397]

[Malawi:Malawi, Zebra, 29223, 2366]
[Malawi:Malawi, Zebra, 29223, 2366]
[Malawi:Malawi, Zebra, 29223, 2366]
[Morocco:Morocco, Zebra, 49929, 6301]
[Morocco:Morocco, Zebra, 49929, 6301]
[Morocco:Morocco, Zebra, 49929, 6301]
[Namibia:Namibia, Zebra, 52332, 4700]
[Namibia:Namibia, Zebra, 52332, 4700]
[Namibia:Namibia, Zebra, 52332, 4700]
[Oregon:Oregon, Zebra, 76772, 8277]
[Oregon:Oregon, Zebra, 76772, 8277]
[Oregon:Oregon, Zebra, 76772, 8277]
[Rhode_Isl:Rhode_Isl, Zebra, 44819, 897]
[Rhode_Isl:Rhode_Isl, Zebra, 44819, 897]
[Rhode_Isl:Rhode_Isl, Zebra, 44819, 897]
[Singapore:Singapore, Zebra, 52432, 4634]
[Singapore:Singapore, Zebra, 52432, 4634]
[Singapore:Singapore, Zebra, 52432, 4634]
[Slovakia:Slovakia, Zebra, 84165, 8774]
[Slovakia:Slovakia, Zebra, 84165, 8774]
[Slovakia:Slovakia, Zebra, 84165, 8774]
[Sudan:Sudan, Zebra, 63852, 4900]
[Sudan:Sudan, Zebra, 63852, 4900]
[Sudan:Sudan, Zebra, 63852, 4900]
[Tonga:Tonga, Zebra, 60491, 6789]
[Tonga:Tonga, Zebra, 60491, 6789]
[Tonga:Tonga, Zebra, 60491, 6789]
[Tunisia:Tunisia, Zebra, 81730, 8873]
[Tunisia:Tunisia, Zebra, 81730, 8873]
[Tunisia:Tunisia, Zebra, 81730, 8873]
[Tuvalu:Tuvalu, Zebra, 2517, 6799]
[Tuvalu:Tuvalu, Zebra, 90204, 4406]
[Tuvalu:Tuvalu, Zebra, 90204, 4406]
[Vatican_C:Vatican_C, Zebra, 72896, 3226]
[Vatican_C:Vatican_C, Zebra, 72896, 3226]
[Vatican_C:Vatican_C, Zebra, 72896, 3226]
Number of maps found: 66
Number of disk page reads: 805
Number of disk page writes: 0
This process took: 991 milliseconds

Buffer Frames: 100
DB: joindb
Order: Type 1
query jointable 1 * Zebra * 100
DB name: jointable
[Alabama:Alabama, Zebra, 50568, 2437]
[Alabama:Alabama, Zebra, 50568, 2437]
[Alabama:Alabama, Zebra, 50568, 2437]
[CZECH_REP:CZECH_REP, Zebra, 60590, 4139]
[CZECH_REP:CZECH_REP, Zebra, 60590, 4139]
[CZECH_REP:CZECH_REP, Zebra, 60590, 4139]
[Colombia:Colombia, Zebra, 63378, 7577]
[Colombia:Colombia, Zebra, 90587, 3494]
[Colombia:Colombia, Zebra, 90587, 3494]
[Dominica:Dominica, Zebra, 86890, 1582]
[Dominica:Dominica, Zebra, 86890, 1582]

[Dominica:Dominica, Zebra, 86890, 1582]
[Georgia:Georgia, Zebra, 84319, 4131]
[Georgia:Georgia, Zebra, 84319, 4131]
[Grenada:Grenada, Zebra, 71893, 1976]
[Grenada:Grenada, Zebra, 71893, 1976]
[Grenada:Grenada, Zebra, 71893, 1976]

[Guatemala:Guatemala, Zebra, 64486, 4702]
[Guatemala:Guatemala, Zebra, 64486, 4702]
[Guatemala:Guatemala, Zebra, 64486, 4702]
[Ireland:Ireland, Zebra, 85673, 4680]
[Ireland:Ireland, Zebra, 85673, 4680]
[Ireland:Ireland, Zebra, 85673, 4680]
[Latvia:Latvia, Zebra, 25255, 4782]
[Latvia:Latvia, Zebra, 25255, 4782]
[Latvia:Latvia, Zebra, 25255, 4782]
[Madagasca:Madagasca, Zebra, 41328, 7397]
[Madagasca:Madagasca, Zebra, 41328, 7397]
[Madagasca:Madagasca, Zebra, 41328, 7397]
[Malawi:Malawi, Zebra, 29223, 2366]
[Malawi:Malawi, Zebra, 29223, 2366]
[Malawi:Malawi, Zebra, 29223, 2366]
[Morocco:Morocco, Zebra, 49929, 6301]
[Morocco:Morocco, Zebra, 49929, 6301]
[Morocco:Morocco, Zebra, 49929, 6301]
[Namibia:Namibia, Zebra, 52332, 4700]
[Namibia:Namibia, Zebra, 52332, 4700]
[Namibia:Namibia, Zebra, 52332, 4700]
[Oregon:Oregon, Zebra, 76772, 8277]
[Oregon:Oregon, Zebra, 76772, 8277]

[Oregon:Oregon, Zebra, 76772, 8277]
[Rhode_Isl:Rhode_Isl, Zebra, 44819, 897]
[Rhode_Isl:Rhode_Isl, Zebra, 44819, 897]
[Rhode_Isl:Rhode_Isl, Zebra, 44819, 897]
[Singapore:Singapore, Zebra, 52432, 4634]
[Singapore:Singapore, Zebra, 52432, 4634]
[Singapore:Singapore, Zebra, 52432, 4634]
[Slovakia:Slovakia, Zebra, 84165, 8774]
[Slovakia:Slovakia, Zebra, 84165, 8774]
[Slovakia:Slovakia, Zebra, 84165, 8774]
[Sudan:Sudan, Zebra, 63852, 4900]
[Sudan:Sudan, Zebra, 63852, 4900]
[Sudan:Sudan, Zebra, 63852, 4900]
[Tonga:Tonga, Zebra, 60491, 6789]
[Tonga:Tonga, Zebra, 60491, 6789]
[Tonga:Tonga, Zebra, 60491, 6789]
[Tunisia:Tunisia, Zebra, 81730, 8873]
[Tunisia:Tunisia, Zebra, 81730, 8873]
[Tunisia:Tunisia, Zebra, 81730, 8873]
[Tuvalu:Tuvalu, Zebra, 2517, 6799]
[Tuvalu:Tuvalu, Zebra, 90204, 4406]
[Tuvalu:Tuvalu, Zebra, 90204, 4406]
[Vatican_C:Vatican_C, Zebra, 72896, 3226]
[Vatican_C:Vatican_C, Zebra, 72896, 3226]
[Vatican_C:Vatican_C, Zebra, 72896, 3226]
Number of maps found: 66
Number of disk page reads: 805
Number of disk page writes: 0
This process took: 751 milliseconds

B.6 Map Count

getcount
DB: dbsort1
Map count: 9777
Distinct row count: 100
Distinct column count: 99
DB: dbsort2
Map count: 9777
Distinct row count: 100
Distinct column count: 99
DB: joindb
Map count: 10275
Distinct row count: 22
Distinct column count: 295
DB: dbtest1
Map count: 25994
Distinct row count: 100
Distinct column count: 99
DB: dbtest2
Map count: 25994
Distinct row count: 100
Distinct column count: 99
DB: dbtest3
Map count: 9777
Distinct row count: 100
Distinct column count: 99