Report for Phase #3:

Implementing Column-Oriented Database Management System

CSE 510: Database Management System Implementation
(Spring 2018)

Date of Submission: April 29, 2018

Group Number 5
Sucharitha Rumesh
Supriyaa Damodaraswamy
Spoorthi Karnati
Naga Subrahmanya Sai Tarun Malladi
Hariharan Adhithya Venkatraman
Thiviyakalyani Navaneethan
Ashish Agarwal

TABLE OF CONTENTS

ABSTRACT

A Database Management System handles a collection of data, their storage, update and access. Joins are among the most expensive operators in a relational database system, and their implementation has a big impact on performance. The join helps you to combine data from two or more tables, thereby reducing the need to repeat columns and columns of data in the same table. In this phase of the project, we aim to implement join operations on Column-Oriented Database leveraging the modules of Minibase for Java. Minibase is a relational Database Management System that is a Row-oriented Database. In Column-store, each column or attribute value is stored contiguously. They perform better on I/O as only the necessary attributes have to be accessed. The implementation covers details such as how different type of joins were implemented, indexing using B+ tree and Bitmap strategies, for joining two tables.

KEYWORDS

Column Oriented Database, duplicate elimination, file scan, index scan, nested loop joins, sort merge join, sorting, ColumnarIndexScan, ColumnarNestedLoopJoins, ColumnarSort, ColumnarBitmapEquiJoins

1 INTRODUCTION

Relational Database Management System is traditionally implemented as a Row-store. When querying is done in such system we have too read the whole record and do a projection of the required attributes. This is an expensive operation when there are thousands of records and only very few attributes being projected. It is also not feasible to use this type of database when the schema is frequently changing, with addition and deletion of attributes. This will require accessing each record to delete the attribute or creating space for each record for the extra attribute.

When presented with such a situation, implementation of the Relational Database as a Column-store is an effective solution. In Column-oriented Database, the records are split into columns and stored in separate columnar files. A specific column data can be accessed contiguously when querying. Each value in a columnar file is accessed using its Tuple id by initiating a Tuple scan. Indexing of Column store is done by extending the BTree index of row store. Here the leaf data page stores the columnar file values.Column store is more effectively indexed using a Bitmap. Each column will have a number of Bitmaps depending on the number of unique values in that column. In Bitmap, a bit is set if the value in the corresponding position in the columnar file is equal to the value on which the bitmap is being created.

In this Phase of the project, we aim to implement different types of joins on our columnar database. In this phase of the project, we have focus on the Iterator class, which provides methods for duplicate elimination, file scan, index scan, nested loop joins, sort merge join, and sorting. In the previous phase, we have already worked on file scan and index scan on columnar files. In this phase, we will generalize these.

2 BACKGROUND

2.1 Terminology:

In this phase of the project, we will focus on the Iterator class, which provides methods for duplicate elimination, file scan, index scan, nested loop joins, sort merge join, and sorting. Following are some of the terminologies that are commonly used during this phase of the project.

Column Oriented Database:

A column-oriented DBMS (or columnar database management system) is a DBMS that stores data tables by column rather than by row. By storing data in columns rather than rows, the database can more precisely access the data it needs to answer a query rather than scanning and discarding unwanted data in rows and hence query performance is often increased as a result in very large data sets.

Relational Algebra:

Relational Algebra is procedural query language, which takes Relation as input and generate relation as output. Relational algebra mainly provides theoretical foundation for relational databases and SQL.

Relational Operators:

Relational Set Operators uses relational algebra to manipulate contents in a database. They are used to combine or subtract the records from two tables. These operators are used in the SELECT query to combine the records or remove the records.

Access Methods:

An access method (or index) facilitates retrieval of a desired record from a relation. In Minibase, all records in a relation are stored in a Heap File. An index consists of a collection of records of the form <key value,rid>, where `key value' is a value for the search key of the index, and `rid' is the id of a record in the relation being indexed. Any number of indexes can be defined on a relation.

File Scan:

A scan on all the columns together to determine if the tuple needs to be selected for join. Works similar to a row-store scan.

B+ Trees:

B+ tree is a (key, value) storage method in a tree like structure having one root, any number of intermediary nodes (usually one) and a leaf node where all leaf nodes will have the actual records stored. Intermediary nodes will have only pointers to the leaf node which does not have any data.

Joins:

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Nested Loop Joins:

The nested loop join, uses one join input as the outer input table and the other input as the inner input table. The outer loop consumes the outer input table row by row. The inner loop, executed for each outer row, searches for matching rows in the inner input table. In a nested-loop join, the database server scans the first, or *outer table*, and then joins each of the rows that pass table filters to the rows found in the second, or *inner table.* A Nested Loop Join is the simplest physical implementation of joining two tables.

Block Nested Loop Joins:

In this method, in addition to looping each record in outer and inner tables, their blocks are also added for looping. Hence, this method selects block of records from both the tables and compares the records in them. Hence the processing is done block wise.

Indexed Nested Loop Joins:

When indexes are used on the columns that are used in join condition, it will not scan each records of inner table. It will directly fetch matching record. But we will have the cost for fetching the index in the index table. Indexes are useful when natural joins are joins or equijoins are used. Hence if indexes are not defined on the columns, we can create temporary indexes to run the query.

Merge Joins:

Tables used in the join query may be sorted or not sorted. Sorted tables give efficient costs while joining. In this method, the column used to join both the tables is used to sort the two tables. It uses merge-sort technique to sort the records in two tables. Once the sorting is done then join condition is applied to get the result. Joining is also similar to any other joins, but if two records have same column value, then care should be taken to sort records based on all the columns of the record. This method is usually used in natural joins or equijoins.

Hash Joins:

This method is also useful in case of natural and equi joins. We use hash function h on the joining column, to divide the records of each tables into different blocks. Assume each of these hash divided block of records fit the memory block and we have NH number of hash divided blocks.

Sort:

A Sort clause is used to order the given tuple in ascending or descending based on the input column.

2.2 Assumptions:

1. The data is not persistent.
2. Only string and integer values are supported.
3. Works with two tables only.
4. First character of the input string will be column name i.e. (A, B, C, D).
5. Second character will be the operator that can be one of these only (>, <, =).
6. All the characters after second will be considered as column value.
7. All conditional expressions are in CNF.
8. ColumnarIndexScan works with the access type BTREE only.
9. Nested loop join works with the access type FILESCAN and BTREE only.
10. The order in which input is given for nested loop join query is in the following order *nlj COLUMNDB OUTERFILE INNERFILE OUTERCONST INNERCONST JOINCONST OUTERACCTYPE INNERACCTYPE [TARGETCOLUMNS] NUMBUF*
11. Bitmap can be created for maximum 3 columns only with only one if it being string type.
12. Bitmap Equijoin query is of the form: bmj COLUMNDB COLUMNARFILENAME1 COLUMNARFILENAME2 [TARGETCOLUMNNAMES] [LEFTCONST] [RIGHTCONST] [EQUICONST] NUMBUF
13. EQUICONST can contain a maximum of two AND in its expression.
14. No Optimizations are performed for join queries and frequent queries.

3 IMPLEMENTATION

3.1 CLASS DECLARATION:

The following is a list of tasks that we have performed for this final phase of the project.

1. Generalize that class to a **ColumnarIndexScan** class, with constructor (as shown below)which scans the given columnar file using the given one or more index files and returns matching tuples.

```
ColumnarIndexScan(
        String queryInput : Stores command line input
        BufferedReader in: BufferedReader object to read command line input
        String[] input : Stores values of queryInput separated by spaces
        String DBName : name of the column database
        String filename :name of the file
        String valueconstraint: stores the complete set of conditions
        String accesstype: stores the complete set of accesstypes

)
```

Methods implemented in this class are:

getFinalTuples(
    ColumnarFile cf:Columnar file object
    String valueconstraint:constraints that has to be applied on table
    String accesstype: accesstypes that has to be applied on columns
)

getResultsofBtree(
    ColumnarFile cf: Columnar file object
    List<String> valueConstraint: list of constraints that has to be applied after creating btree indexes on columns
    String accessType: the accesstype that has to be applied on the columns
    String columnarFileName: Name of the columnar file

)

createBtreeIndex(
    ColumnarFile cf: Columnar file object
    String columnDbName: Name of the Column DB
    String columnarFileName: Name of the Columnar file
    String columnName: Name of the Column on which index has to br created
)

insertRecords(
    String table : name of the table
)

2. Implement a **ColumnarNestedLoopJoins** class, with *constructor*

ColumnarNestedLoopsJoins (
    String columnDB : name of the column database
    String outerFile : name of the outer data file
    String innerFile : name of the inner data file
)

and *methods* as follows

getResults (
    ColumnarFile outerCf : Columnar file object for outer table
    ColumnarFile innerCf : Columnar file object for inner table
    String outerFile : Outer table filename
    String innerFile : Inner table filename
    String outerConstraint : Constraints to be applied on outer table
    String innerConstraint : Constraints to be applied on inner table

```
        String joinConstraint : Constraint on which table has to be joined
        String outerAccessType : Access type for the outer table
        String innerAccessType : Access type for the inner table
        List<String> targetColumnNames : List of the output columns
        int numOfBuffers : Maximum number of buffers to be used
)

getConditionList(
        String tableCondition : conditional expression as a string (single condition only)
)

getColumnNumber(
        String columnName : name of the column
)

getResultsFileScan(
        ColumnarFile cf : columnar file instance
        List<String> targetColumnNames : list of target columns to be fetched
        List<String> valueConstraint : conditional expression as list
        String accessType : access type for fetching records, FILESCAN here
        int numBuf : Max number of buffers allowed
)

createBtreeIndex(
        ColumnarFile cf : coulmnar file instance
        String columnDbName : name of the columnar database
        String columnarFileName : name of the columnar file
        String columnName : name of the column on which index has to be created
)

getResultsBtree(
        ColumnarFile cf : columnar file instance
        List<String> targetColumnNames : list of the target columns to be fetched
        List<String> valueConstraint : conditional expression as list
        String accessType :  access type for fetching records, BTREE here
        int numBuf : Max number of buffers allowed
        String columnarFileName : name of the columnar file for index creation
)

fetchResultsFromPosition(
        ColumnarFile cf : columnar file instance
        int position : position from where records needs to be fetched
)
```

3. Implement a BitmapEquiJoin class, with a <u>constructor,</u>

BitmapEquiJoin(
    AttrType[] input1: Attribute types of columns in left columnar file
    int length_in1: number of columns in left columnar file

    short[] t1_sizes: size of strings in left columnar file
    AttrType[] input2: Attribute types of columns in right columnar
    int length_in2: number of columns in right columnar file
    short[] t2_sizes: size of strings in left columnar file,
    int numbuf: number of buffers
    String leftFileName: Left columnar file name
    String righFileName Right columnar file name
    FldSpec[] list: relations to be used
    int n_out: Number of output fields

)

and <u>methods</u> as follows,

MatchLeftCondition(
    Map bitmapMeta: Hashmap containing metadata for bitmap
    CondExpr Left: head of linked list for outer conditional expression
)

MatchRightCondition(
    Map bitmapMeta: Hashmap containing metadata for bitmap
    CondExpr Right: head of linked list for inner conditional expression
)

MatchingValues(
    Map bitmapMeta : Hashmap containing metadata for bitmap
    Map Join: Hashmap containing matching positions on left and right columnar file
    CondExpr Equi: head of linked list for equi conditional expression
    TreeSet<Integer> Left: Positions matching Left Const
    TreeSet<Integer>Right: Positions matching Right Const
)

getColumnNumber(String columnName: Column name to find number for)

getColumnName(int columnNum: Column Number to find column name for)

4. Implement a **ColumnarSort** class with constructor (as shown below) which sorts the given columnar file based on the given conditions,

   ColumnarSort(
   AttrType[] in : Array of Attribute types,
   short len_in : No. of columns in the relation,
   short[] str_sizes : Array of sizes of the string attributes present in the relation,
   java.lang.String ColumnarFileName : Name of the columnar file,
   int sort_fld : Number of the field to sort on,
   TupleOrder sort_order : Order in which the sort has to be performed,
   int sort_fld_len : Length of the sort field,
   int n_pages : Memory available for sorting (in pages),
   ColumnarFile cf : Columnar file instance in which we work on)

   and methods:

   get_next()

   generate_runs(
   int max_elements: Maximum elements that can be present in the run queue,
   AttrType sortFldType : Attribute type of the field to be sorted on,
   int sortFldLen: Length of the field to be sorted on)

   delete_min()

   setup_for_merge(

   int tuple_size : Size of each tuple,

   int n_R_runs : Number of runs required that is received from the generate_runs)

   copy_tuple(

   Tuple tuple : The tuple which has to be copied to the new tuple)

   KWayMerge(

   List heapfiles : List of the runs that has been put into a heapfile,

   List<Integer> currHeapFileRecCntList : The list of chunks that is received from the chunk_list function for every heapfile,

   Heapfile currHeapFile : The current heap file in which we work currently)

3.2 MEMBER FUNCTION IMPLEMENTATIONS:

3.2.1 Columnar Index Scan

ColumnarIndexScan(String queryInput,BufferedReader in,String[] input,String DBName,String filename,String valueconstraint,String accesstype)

This is a parameterized constructor of the ColumnIndexScan class. The parameters are initialized in this method.

getFinalTuples(cf,valueconstraint,accesstype)

This method gets the final tuples that matches the given set of CNF conditions. Firstly, the given constraint value and index values are parsed and then sent to another method where the tuples of each atomic constraint are calculated.

getResultsofBtree(cf,andcond, indexType.get(i),"file")

This method returns the tuples which are matching with the atomic condition with BTREE index created on the column on which the constraint is declared.

getResultsofBitmap(cf,andcond, indexType.get(i))

This method returns the tuples which are matching with the atomic condition with BITMAP index created on the column on which the constraint is declared.

3.2.2 Columnar Nested Loop Joins

ColumnarNestedLoopsJoins(String columnDB, String outerFile, String innerFile)

This is a parameterized constructor of the class. It instantiates the class object with few of the member variables values initialized.

getResults(
ColumnarFile outerCf, ColumnarFile innerCf,
String outerConstraint, String innerConstraint, String joinConstraint,
String outerAccessType, String innerAccessType,
List<String> targetColumnNames, int numOfBuffers):

This is the function that is used to fetch the join results from the two table. This function then converts the input outer constraint, inner constraint, join constraint into standard format. Further it splits the conditions on the and (&) operator. Then on the basis of accesstype of outer and inner tables we fetch the results. Results are stored in such a way that whenever there is and condition we take intersection of results and whenever there is union we take union of results. Index and other things are also taken care of under the individual access types. Description of that can be found under different method description below.

getConditionList(String tableCondition)

This function takes single condition as an input. It parses this condition and creates a list which contains the name of the column, operator and the value that is used for comparison. Here, we have assumed that the first character of the input string will be column name i.e. (A, B, C, D), second character will be the operator that can be one of these only (>, <, =) and all the characters after second will be considered as value. The function returns the list having 3 elements as column name, operator.

getColumnNumber(String columnName)

This function is used to get the column number corresponding to the column name, provided as an function parameter. The function uses switch case statement on the input column name string (here A, B, C or D) and returns the corresponding column number as (1, 2, 3 or 4). This function is required when we have requirement of fetching the values from the list of results then this column number returned can be treated as index.

**getResultsFileScan(ColumnarFile cf, List<String> targetColumnNames,
List<String> valueConstraint, int numBuf)**

This function is called when the selected access type is chosen as FILESCAN. This function takes the required columnar file object, target columns, single constraint and the number of buffers as a input. The function first initialises the tuple scan on the given columnar file, it also defines the tuple id variable with required parameters. Further, it creates a key based on the type of our column which will be used for comparison during the scanning. Once done, this function starts scanning tuples one by on and based on the operator value it either selects the row or move to fetch next tuple. Once rows satisfying the required condition is found we store then in a list of output results and return.

**createBtreeIndex(ColumnarFile cf, String columnDbName,
String columnarFileName, String columnName)**

This function creates the btree index on the given column. Function takes the columnar file instance and column name on which index has to be created. We open the heapfile corresponding to the given column and start creating the btree index file on the basis of the column attribute type. Once btree index file is created we start traversing the column heap file and insert the column values into a btree index file.

**getResultsBtree(ColumnarFile cf, List<String> targetColumnNames,
List<String> valueConstraint, int numBuf, String columnarFileName)**

This function is called when the selected access type is chosen as BTREE. This function takes the required columnar file object, target columns, single constraint and the number of buffers and whether the index is on outer file or inner file as a input. First using the constraint list we fetch the column on which index needs to be created. After that corresponding heapfile is opened that will be used to find the actual record values using the indexes. Further, it creates a lowkey and a highkey based on the type of our column which will be used for comparison during the scanning. Once done, this function starts the index scan using the index file of the column and start scanning tuples one by on and based on the operator value it either selects the row or move to fetch next tuple. Once rows satisfying the required condition is found we find the position of the particular record id and then the corresponding results are fetched using the columnar file and stored in a list of output results. Finally, the list of output results is returned.

fetchResultsFromPosition(ColumnarFile cf, int position)

This function takes the columnar file and the position of the record that is find when we are using btree as our access type. This function then converts the position to the corresponding rid and actual values are fetched for all the columns having this rid. Finally the result is returned.

<u>3.2.3 BitmapEquiJoin</u>

BitmapEquiJoin(AttrType[] input1, int length_in1, short[] t1_sizes ,AttrType[] input2, int length_in2,short[] t2_sizes, int numbuf, String leftFileName, String righFileName, FldSpec[] list, int n_out)

This is a parameterized constructor that initializes the output fields information and the metadata of the left and right columnar file.

MatchLeftCondition(Map bitmapMeta, CondExpr Left)

This function takes the bitmap Metadata and the Conditional expression on the left columnar file as the inputs. It first checks the operator type of Left. If it is “equal to” then the correspond bitmap file that matches the condition is opened and positions are fetched and stores in a TreeSet which is returned. Otherwise if the operator is “less than” then all bitmaps with value less than the given condition value are opened and positions in each one is fetched and stored to the TreeSet which is returned. Similarly for operator “greater than” all bitmaps with values greater than the given condition value are opened and positions in each one is fetched and stored to the TreeSet which is returned. This fetching position operation is done for each condition in the linked list of conditions.

MatchRightCondition(Map bitmapMeta, CondExpr Right)

This function takes the bitmap Metadata and the Conditional expression on the right columnar file as the inputs. It first checks the operator type of Right. If it is “equal to” then the correspond bitmap file that matches the condition is opened and positions are fetched and stores in a TreeSet which is returned. Otherwise if the operator is “less than” then all bitmaps with value less than the given condition value are opened and positions in each one is fetched and stored to the TreeSet which is returned. Similarly for operator “greater than” all bitmaps with values greater than the given condition value are opened and positions in each one is fetched and stored to the TreeSet which is returned. This fetching position operation is done for each condition in the linked list of conditions.

MatchingValues(Map bitmapMeta, Map Join, CondExpr Equi, TreeSet<Integer> Left, TreeSet<Integer>Right)

Using the equi condition, find the left and right join field. Get the unique values in those columns using the bitmapMeta. Find the values that are common to both the columns, these will be the values on which join is done. For each of these values open the bitmaps for left and right files and fetch positions. if the fetched positions are also present in the corresponding Left or Right TreeSet the add them to Left_Positions or Right_Positions list correspondingly . If both lists have elements then add them to Join. Repeat this process for each unique value in the intersection set and for each condition in the linked list of conditions.

getColumnNumber(String columnName)

This function is used to get the column number corresponding to the column name, provided as an function parameter. The function uses switch case statement on the input column name string (here A, B, C or D) and returns the corresponding column number as (1, 2, 3 or 4). This function is required when we have requirement of fetching the values from the list of results then this column number returned can be treated as index.

getColumnName(int columnNum)

This function is used to get the column name corresponding for the column number, provided as an function parameter. The function uses switch case statement on the input column number (here 1, 2, 3 or 4) and returns the corresponding column name as (A, B, C or D). This column name is required when a key contains the column name as a part.

<u>3.2.4 ColumnarSort</u>

ColumnarSort(AttrType[] in, short len_in,short[] str_sizes, java.lang.String ColumnarFileName,int sort_fld,TupleOrder sort_order,int sort_fld_len,int n_pages,ColumnarFile cf)

This is a parameterized constructor that initializes the member variables and get the meta data of the tuple and initialises the filescan with help of the given heapfile and other required parameters. This filescan object is used in initializing the sort iterator that is used for ordering the given tuple based on the order and column given.

get_next()

This function helps in getting the next tuple that is in the sort order given and using it, we get the position that is fixed in the second field of it. Using the position that is received, we project the position for all the other columns and give the result as a whole tuple.

generate_runs(int max_elements, AttrType sort_field, int sortFldLen)
This function helps in getting the number of runs required for the setup for merge. It initially assigns the minimum value to the lastElem. Then it uses two queues that are implemented as a sorted binary tree. One queue is to enqueue those tuples that are good in state ,i.e., if the current element is greater than or equal to the lastElem. Else, it is enqueued to the other queue which represents the inactive state. It runs until the active queue becomes equal to the max_elements. If it is full, the queue is dequeued and written into the disk and whenever there is a free space in the active queue, the other elements from the run is pushed into it. Once the inactive queue becomes full, a new run is started and the same process is repeated. Once there are no more elements left, the run is returned.

setup_for_merge(int tuple_size, int n_R_runs)
The function gets the size of the tuple and the number of runs from the generete_runs and merges the all the runs with the help of the KWayMerge which is optimized in solving the merge in linear time without much dist access. It gets the elements in ascending order and put it into the final Q in the same order which is used later with the delete_min to get the minimum values in order.

delete_min()
It helps in removing the root of the final queue that has all the elements after the merge.

copy_tuple(Tuple tuple)
It gets the tuple that has size one and copies it to another tuple of size 2 and returns it.In this tuple, one more attribute is added for storing the position of that tuple which helps in the projection.

<u>3.2.5 Test Driver Member Function</u>

insertRecords(String table)

We are calling insertRecords() function that reads the header of the given input file and parse the given input i.e. determines the data type of the columns, size (if string) and calls the columnar file constructor to create columnar file object. Once the columnar file object is created the file is further read and data is inserted into the corresponding column heapfile. Further disk reads and writes are calculated in the complete procedure. After successful insertion of records, using tuple scan class all the inserted records were printed and checked whether they were inserted correctly or not. For this tuple scan object for created. Tuple metadata was set and using this tuple scan class getNext() function records were fetched and displayed using the predefined Tuple class functions.

PrintRecords(Map<List<Integer>,List<Integer>>Pos, List<String> targetColumnNames, int numcols)

Get the target column numbers. Get the two lists of positions .Using positions get the records using fetchRecordsforgivenposition().Iterate over the Left and Right Records and print the output. Repeat for all Key,Value pairs in Pos.

fetchResultsFromPosition(ColumnarFile cf, int position)

This function takes the columnar file and the position of the record that is find when we are using btree as our access type. This function then converts the position to the corresponding rid and actual values are fetched for all the columns having this rid. Finally the result is returned.

get_Condexpr(ColumnarFile cf, String value)

Split the string value with '|' as the delimiter. Split each part with the relational operator as the delimiter. part at position zero is the string column name which is operand 1 and part at position one is the operand 2. The operator value is also set by comparison. Create a linked list with this expression. Return the linked list.

get_EquiCondexpr(String value)

Split the string value with '|' as the delimiter. Split each part with the relational operator as the delimiter. part at position zero is the string column name which is operand 1 and part at position one is also a string column which is operand 2. The operator is always 'equal to'.

3.3 TEST DRIVER IMPLEMENTATIONS:

3.3.1 Columnar Nested Loop Joins Test

- Query of the format : "query COLUMNDBNAME COLUMNARFILENAME [TARGETCOLUMNNAMES] [VALUECONSTRAINT NUMBUF] ACCESSTYPE".
- An object of the class ColumnarIndexScan() is created to which in turn will perform all columnar index scan function of the given query.

3.3.2 Columnar Nested Loop Joins Test

- Query of the format : "nlj COLUMNDB OUTTERFILE INNERFILE OUTERCONST INNERCONST JOINCONST OUTERACCTYPE INNERACCTYPE [TARGETCOLUMNS] NUMBUF".

- The data from both outer and inner file are fetched and inserted into a columnar file "outer" and "inner" columnar file respectively by using the insertRecords(String Table) function.
- The TARGETCOLUMNS provided in the query are fetched and inserted into an array.
- The Function type from the query is checked, if it it "nlj", a ColumnarNestedLoopJoins object is created.
- The result of the columnar nested loop join is done by using the function getResult and stored in a object "result" which is of the type List which contains a list of strings.
- This list is put into a for loop to display the final result.

<u>3.3.3 Bitmap Equality Joins Test</u>

- Query of the format : "bmj COLUMNDB OUTTERFILE INNERFILE [TARGETCOLUMNNAMES] [LEFTCONST] [RIGHTCONST] [EQUICONST] NUMBUF".
- The data from both outer and inner file are fetched and inserted into a columnar file "outer" and "inner" columnar file respectively by using the insertRecords(String Table) function.
- A do-while loop is created to ask the user if they want to create a BITMAP index on a particular column in a particular file.
- The BITMAP index for all columns are created for both the inner as well as the outer file.
- The TARGETCOLUMNS provided in the query are fetched and inserted into an array.
- The LEFTCONST is split by '&' each part of split is used to create a conditional expression linked list with split on '|'. These conditional expressions are then stored in a List . Similarly RIGHTCONST and EQUICONST are also made into to lists.
- Iterating over the LeftConst list call the MatchLeftCondition() and store the unique returned results to a TreeSet Left.
- Iterating over the RightConst list call the MatchRightCondition() and store the unique returned results to a TreeSet Right.
- We assume there is a maximum of 3 conditions for EquiConst. Call MatchingValues() with the bitmap metadata and the Tree Sets from above steps as parameter. The fourth parameter is changed every time with Join1, Join2 and Join3 .
- Iterate over these three sets and join them to get final positions on left and right columnar file.
- Using fetchResultsfromposition() get the record value and Print the records using PrintRecord().

4 PERFORMANCE ANALYSIS

Following is the table that shows the total number of disk reads and writes that were performed for different scenarios:

Columnar Index Scan

Number of buffers used: 5

Query:
query sample_data [(A=New_Hampshire|C=8)&D=6&C=6] BTREE

ColumnarIndexScan	Disk Reads	Disk Writes
	14	2

Column Nested Loop Join

Number of buffers used: 5

Query:
nlj column smalldata_part1 smalldata_part2
[(A=New_Hampshire|C=8)&D=6&C=6] [(A=Montana|C=8)&D=6] [C=D]
BTREE FILESCAN [A,B,C,D] 5

Insert Queries	Disk Reads Outer File	Disk Writes Outer File	Disk Writes Inner File	Disk Writes Inner File
	14	39	18	89

Nested Loop Joins Access Type	Disk Reads	Disk Writes
Outer: FILESCAN Inner: FILESCAN	819	278

Nested Loop Joins Access Type	Disk Reads	Disk Writes
Outer: BTREE Inner: FILESCAN	5343	321

Nested Loop Joins Access Type	**Disk Reads**	**Disk Writes**
Outer: FILESCAN Inner: BTREE	3549	303

Nested Loop Joins Access Type	**Disk Reads**	**Disk Writes**
Outer: BTREE Inner: BTREE	5708	569

Bitmap Equi join

Number of buffers used: 5

Query:
bmj column smalldata_part1 smalldata_part2 [A]
[(A=New_Hampshire|C=8)&D=6&C=6] [(A=Montana|C=8)&D=6] [C=D] 5

Bitmap Equi Join	**Disk Reads**	**Disk Writes**
	99	186

5 INTERFACE SPECIFICATIONS:

- Data is stored in Internal memory
- The java core program provides a command line interface to enter the query.
- Eclipse IDE for development and testing.

6 SYSTEM REQUIREMENTS/INSTALLATION AND EXECUTION INSTRUCTIONS:

6.1 Software requirements:

1. JDK 1.8
2. JVM
3. Eclipse

6.2 Hardware Minimum Requirements:

1. 4GB + of RAM memory
2. 1 GB of storage space
3. 32 bit processor

6.3 Installation and Execution Instructions :

Download the Zip folder and unzip it. Two data sets for the two columnar files are stored in two text files. The file path in the test driver has to be modified to reflect the path of the text files.

Columnar Index Scan:

```
query sample_data [(A=New_Hampshire|C=8)&D=6&C=6] BTREE
```

Column Nested Loop Join:

```
nlj column smalldata_part1 smalldata_part2
[(A=New_Hampshire|C=8)&D=6&C=6] [(A=Montana|C=8)&D=6] [C=D]
BTREE FILESCAN [A,B,C,D] 5
```

Bitmap Equi join:

```
bmj column smalldata_part1 smalldata_part2 [A]
[(A=New_Hampshire|C=8)&D=6&C=6] [(A=Montana|C=8)&D=6] [C=D] 5
```

Columnar Sort:

```
sort c c C ASC 20
```

Ascending order: ASC
Descending order: DSC

7 RELATED WORK

TEXT	AUTHOR	DESCRIPTION
Chapter 12: Overview Of Query Evaluation [Database Management Systems by Raghu Ramakrishnan]	Raghu Ramakrishnan	We get an overview of how queries are evaluated in a relational DBMS. We get to know how SQL queries are translated into an extended form of relational algebra, and how query evaluation plans are represented as trees of relational operators, along with labels that identify the algorithm to use at each node. Finally, we learned that how relational operators serve as building blocks for evaluating queries, and the implementation of these operators is carefully optimized for good performance.
Chapter 14: Evaluating Relational Operators [Database Management Systems by Raghu Ramakrishnan]	Raghu Ramakrishnan	In this chapter, we studied the implementation of individual relational operators in sufficient detail to understand how DBMS are implemented. Joins are among the most expensive operators in a relational database system, and their implementation has a big impact on performance. We consider implementation of the binary operators cross-product, intersection, union, and set-difference

8 CONCLUSION

From this project we have gained a comprehensive understanding of how column oriented databases work. We learned about the basic design principles that must be taken into consideration while designing a database. Also, we gained a hands-on experience on an open source row oriented database called, Minibase. We learned about the different modules of databases and how they are implemented in MiniBase and further used these as building blocks for implementing a column-oriented DBMS. This course as well as project provided us a platform to understand and work on basic concepts of databases.

9 FUTURE WORK

- Value class can be extended further to support other data types also.
- The data could be made persistent and the scope can be increased beyond the session.
- Support for >=, <=, != conditions can be added to value constraints.
- For nested loop joins bitmap can also be chosen as access type.

10 ACKNOWLEDGMENTS

We would like to take opportunity to express our humble gratitude to Dr. K. Selcuk Candan who motivated and guided us to execute this project. His constant guidance and willingness to share his vast knowledge made us understand this project and its manifestations in great depths and helped us to work on the tasks assigned to us.

We are highly thankful to our project internal guide Ashish Gadkari whose invaluable guidance helped us understand the project better.

Although there may be many who remain unacknowledged in this humble note of gratitude there are none who remain unappreciated.

11 REFERENCES/BIBLIOGRAPHY

[1] Column Oriented DBMS

[2] The Minibase Home Page

[3] https://www.geeksforgeeks.org/

[4] https://www.tutorialcup.com/dbms/

[5] CSE510 - Database Management System Implementation (S18)

[6] Database Management Systems by Raghu Ramakrishnan

[7] https://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.html

[8] http://db.csail.mit.edu/projects/cstore/abadi-sigmod08.pdf

[9] https://en.wikipedia.org/wiki/Column-oriented_DBMS

[10] The Minibase Homepage http://research.cs.wisc.edu/coral/minibase/minibase.html

[11] http://www.cs.carleton.edu/faculty/dmusican/cs334f07/proj1.html

[12] http://www.cs.colostate.edu/~iray/teach/cs430/BufMgr.pdf

[13] https://www.eecs.yorku.ca/course_archive/2003-04/W/4411/proj/smj/

[14] http://www.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/Chap7/node11.html

[15] https://www.tutorialcup.com/dbms/index.html

[16] https://web.stanford.edu/class/cs346/2015/notes/Lecture_One.pdf

12 APPENDIX

Following was the task distribution:

Sucharitha Rumesh	Test Driver, Report
Supriyaa Damodaraswamy	Test Driver, Report
Spoorthi Karnati	Columnar Index Scan, Report
Naga Subrahmanya Sai Tarun Malladi	Columnar Sort, Report
Hariharan Adhithya Venkatraman	Columnar Sort, Report
Thiviyakalyani Navaneethan	Columnar Bitmap EquiJoins, Report
Ashish Agarwal	Columnar Nested Loop Joins, Report