Report Document for **Phase #2:**

"Implementing a Column-Oriented Database Management System."

CSE 510: Database Management Sys Implementation (2018 Spring)

Group Members:

Ankit Kadam (1212415800)
Pavan Omanwar (1212452369)
Saumya Priya (1213066723)
Shivam Agarwal(1213238362)
Mahima Gupta (1213110741)
Pranjal Pangaonkar (1213195410)
Prashant Gonarkar (1213149078)

Abstract:

The traditional form of storing information in relational databases has been row store which has existed for a few decades. In order to enhance query processing and data retrieval, a new form of data storage has evolved which is the the column store method. In this column store method, data is stored in columnar files instead of the traditional row storage. This change has lead to huge impacts on the performance of different types of queries and workloads. The given phase uses the modules of MiniBase as building blocks for implementing a column-oriented DBMS where we implement a relational database management system using the columnar structure. The implementation covers details such as insertion, indexing using B+ tree and Bitmap strategies, querying and deletion.

Keywords :

column oriented database, columnar, Scan, Heap, Heap File, B-Trees, B+ Trees, Buffer Management, Catalog, Chain Exception, Disk Manager, Global, Index, Iterator, Javadoc, Lib,

1 Introduction

1.1 Terminology :

Heap:

A Heap is a specialized tree-based data structure satisfying the *heap property.*

Heap Property :

If P is a parent node of C, then the the *value* of P is either greater than or equal to (*in a max heap*) or less than or equal to (*in a min heap*) the value of C. The node at the "top" of the heap (with no parents) is called the *root* node.

Heap File:

In Heap files, records are inserted at the end of the file as and when they are inserted and there is no sorting or ordering of the records. Once the data block is full, the next record is stored in the new block which does not be the very next block. This method can select any block in the memory to store the new records.

B-Trees:

B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. It is a generalization of a binary search tree in that a node can have more than two children. Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data.

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.

Buffer Management:

The buffer manager is the software layer that is responsible for bringing pages from physical disk to main memory as needed. The buffer manages the available main memory by dividing the main memory into a collection of pages, which we called as buffer pool. The main memory pages in the buffer pool are called frames.

The goal of the buffer manager is to ensure that the data requests made by programs are satisfied by copying data from secondary storage devices into buffer. Infact, if a program performs reading from existing buffers. similarly, if a program performs an output statement: it calls the buffer manager for output operation - to satisfy the requests by writing to the buffers.

Catalog:

The database catalog of a database instance consists of metadata in which definitions of database objects such as base tables, views, synonyms, value ranges, indexes, users, and user groups are stored.

Disk Manager:

The disk space manager hides details of the underlying hardware and allows higher levels of the software to think of the data as a collection of pages.
It is the lowest level of software in the DBMS architecture, with manages space on disk. In short, the disk space manager supports the concept of a page as a unit of data, and provides commands to allocate or deallocate a page and read or write a page. the size of a page is chosen to be the size of a disk block and pages are stored as disk blocks so that reading or writing a page can be done in one disk Input/Output.

It is often useful to allocate a sequence of page as a contiguous sequence of blocks to hold data that is frequently accessed in sequential order This capability is essential for exploiting the advantages of sequentially accessing disk blocks. Such a capability, if desired, must be provided by the disk space manager to higher - level layers of the DBMS.

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.

File Scan:

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

Column Scan:

A scan on just one column of the table to determine if the position needs to be selected for retrieval or deletion.

Bitmap:

Using the bitmap index created for a value and a column to return records that are needed to be printed or deleted.

TID and RID:

TID and RID are unique values that identify a tuple and a row respectively, in the database. A TID is a collection of records in which different rows are stored as an array of their RIDs.

The words method and function are used interchangeably throughout the report.

1.2 Goal description :

Implementing a column-oriented Database management system using the modules of MiniBase as building blocks.

The current phase required us to perform various modifications and implementations of the below classes :

a. Creation of a new abstract class- "ValueClass" extending java.lang.Object.
b. Creation of a new tuple ID (TID) class extending java.lang.Object with a global attribute TID and fields as int numRIDs; int position; RID[] recordIDs;
    i. The class will be having the following methods :
        1. void copyTid(TID tid) : To make a copy of the given tid.
        2. boolean equals(TID tid): It Compares two TID objects and returns if they are equal or not.
        3. void writeToByteArray(byte[] array, int offset): It writes the tid into a byte array at offset
        4. void setPosition(int position): It sets the position attribute with the given value in position
        5. void setRID(int column, RID recordID): It sets the RID of the given column
c. Creation of a new package- "columnar" with the following specifications:
    i. Creation of a new class- "Columnarfile"
    ii. Field Summary: static int numColumns, AttrType[] type
    iii. The constructor summary is defined as follows :

Columnarfile(java.lang.String name, int numColumns, AttrType[] type) Initialize: if columnar file does not exits, create one heapfile (‘‘name.columnid’’) per column; also create a ‘‘name.hdr’’ file that contains relevant metadata

iv. The class will be having the following methods :
    1. void deleteColumnarFile(): It deletes all relevant files from the database.
    2. TID insertTuple(byte[] tuplePtr): It insert tuple into file, return its tid
    3. Tuple getTuple(TID tid): It reads the tuple with the given tid from the columnar file
    4. ValueClass getValue(TID tid, column): It Reads the value with the given column and tid from the columnar file.
    5. int getTupleCnt(): It returns the number of tuples in the columnar file.
    6. TupleScan openTupleScan(): It initiates a sequential scan of tuples.
    7. Scan openColumnScan(int columnNo): It initiates a sequential scan along a given column.
    8. boolean updateTuple(TID tid, Tuple newtuple): It updates the specified record in the columnar file.
    9. boolean updateColumnofTuple(TID tid, Tuple newtuple, int column) : It updates the specified column of the specified record in the columnar file
    10. boolean createBTreeIndex(int column): if the given column doesn’t exist, create a BTree index for it.
    11. boolean createBitMapIndex(int columnNo, valueClass value): If it doesn’t exist, create a bitmap index for the given column and value.
    12. boolean markTupleDeleted(TID tid): Add the tuple to a heapfile tracking the deleted tuples from the columnar file.
    13. boolean purgeAllDeletedTuples(): It merge all deleted tuples from the file as well as all from all index files.

v. Creation of a class- “TupleScan” which scans all columns simultaneously to return complete tuples.
    1. The class will be having the following methods :
        a. void closetuplescan(): Closes the TupleScan object
        b. Tuple getNext(TID tid): Retrieve the next tuple in a sequential scan
        c. boolean position(TID tid): Position all scan cursors to the records with the given rids.

d. Creation of a new package called- “bitmap”
    i. Creation of a class for bitmaps called BM
    ii. Method printBitMap(bitmap.BitMapHeaderPage header) is defined for debugging.

iii. Creation of a class called BitMapFile.
   1. Constructors defined for the class are as follows:
      a. BitMapFile(java.lang.String filename) BitMapFile class; an index file with given filename should already exist, then this opens it.
      b. BitMapFile(java.lang.String filename, Columnarfile columnfile, int ColumnNo, valueClass value) BitMapFile class; an index file with given filename should not already exist; this creates the BitMap file from scratch
   2. The class will be having the following methods :
      a. void close(): to close the BitMap file.
      b. void destroyBitMapFile(): to destroy the entire BitMap file.
      c. bitmap.BitMapHeaderPage getHeaderPage(): Access method to data member.
      d. boolean Delete(int position): sets the entry at the given position to 0.
      e. boolean Insert(int position) set the entry at the given position to 1.

iv. Extending of the class HFPage with the following methods:
   void setCurPage_forGivenPosition(int Position) sets the value of curPage to the page which contains the entry at the given position

v. Creation of a class- "BMPage"
   1. The constructor is defined as:
      a. BMPage(): Default constructor
      b. BMPage(Page page): open a BMPage and make this BMpage point to the given page
   2. The class will be having the following methods :
      a. int available_space(): Returns the amount of available space on the page.
      b. void dumpPage(): Dump contents of a page.
      c. boolean empty(): Determining if the page is empty.
      d. void init(PageId pageNo, Page apage): Constructor of class BMPage initialize a new page.
      e. void openBMpage(Page apage) Constructor of class BMPage open a existed BMPage.
      f. PageId getCurPage()
      g. PageId getNextPage()
      h. PageId getPrevPage()
      i. void setCurPage(PageId pageNo): sets value of curPage to pageNo.

j. void setNextPage(PageId pageNo): sets value of nextPage to pageNo
k. void setPrevPage(PageId pageNo): sets value of prevPage to pageNo.
l. byte[] getBMpageArray() void writeBMPageArray(byte[]).

e. Creation of a new class called ColumnDB under diskmgr
f. Create a class called iterator.ColumnarFileScan by modifying the class iterator.FileScan
   i. The constructor is defined as:
      ColumnarFileScan(java.lang.String file_name, AttrType[] in1, short[] s1_sizes, short len_in1, int n_out_flds, FldSpec[] proj_list, CondExpr[] outFilter)
g. Extend the class global.IndexType with BitMapIndex type.Creation of a class called index.ColumnIndexScan by modifying the class index.IndexScan.
   i.
h. Modification of Minibase disk manager such that it counts the number of reads and writes. This leads to addition of pcounter.java to:

```
package diskmgr;
    public class PCounter {
    public static int rcounter;
    public static int wcounter;
    public static void initialize() { rcounter =0; wcounter =0; }
    public static void readIncrement() { rcounter++; }
    public static void writeIncrement() { wcounter++; }
}
```

And modification of read page() and write page() methods of the diskmgr to increment the appropriate counter upon a disk read and write request.

i. Implementation of a program batchinsert. Given the command line invocation
   i. Batchinsert Datafilename Columndbname Columnarfilename Numcolumns
   ii. If the database/columnarfile already exists in the database, the tuples will be inserted into the existing table. At the end of the batch insertion process, the program should also output the number of disk pages that were read and the number of disk pages that were written.

j. Implement a program index. Given the command line invocation
   i. Index Columndbname Columnarfilename Columnname Indextype
   ii. The program should also output the number of disk pages that were read and the number of disk pages that were written

k. Implement a program query. Given the command line invocation
    i. Query Columndbname Columnarfilename Targetcolumnnames] Valueconstraint Numbuf Accesstype
    ii. The program should also output the number of disk pages that were read and the number of disk pages that were written (if any)

l. Implement a program delete query which works like query, but eliminates all matching tuples from the database. The program should also output the number of disk pages that were read and the number of disk pages that were written (if any)

m. Implement a program delete query which works like query, but eliminates all matching tuples from the database. In addition to having all the inputs of query, the input to delete query also specifies whether the deleted tuples will be purged from the database or not

1.3 Assumptions:

1. The value constraints in query contains only one column.
2. The data is not persistent
3. There is no update in BitMap index files.
4. No Optimizations are performed for join queries and frequent queries.
5. Only works for the given database name and wont work for variable database names

2. Description of the proposed solution/implementation:

The solution which has been implemented in this phase of the project is to make a columnar store database which consists of a database which stores multiple tables and each of those tables have one or multiple columns.
The project forms a modularized component architecture, where the modules have specific functionalities of their own but are still dependent on each other.
Following are the main modules of the project:

1. Columnar
2. Data Definition: Its the data class that is used to represent newly introduced types.
3. BitMap: The bitmap module is used for bitmap indexing.
4. Test Driver: It is a console that is used to interact with the columnar database.

To implement the storage of columns, we have used a heapfile.
The Columnarfile class implements the tables in which we store the column data through creating an array of heap files.

The ColumnarMetaFile class contains the metadata for the tables implemented in the Columnarfile class.

2.1 Columnar File Class:

Columnarfile(String name, int numColumns, AttrType[] type):
This is the constructor for the Columnarfile. It takes in three different parameters: a String variable name, and Integer numColumns and an AttrType class object type. The variable name contains the name of the column, numColumns contains the number of columns and type contains the type of information. For each column, we create a separate HeapFile. We iterate over these columns and create a new Heapfile object on every index. The Heapfile object is named as a concatenation of the input name and the current index while we loop through it. We also set the tuple length in the ColumnarMetaFile class with the setTupleLength method. This method sums up the length of the AttrType fields to define the length of the tuple.

deleteColumnarFile():
We loop through the heapfiles using the heapfile array and call the deleteFile() method of the Heapfile class. This method deletes the entire columnar file as the file is consists of numerous heapfiles. We delete the metadata in the same way.

pushMetaInfo(int string_length):
The pushMetaInfo function is used to insert metadata information in the columnarMetaFile which is an object of the class of the same name. We define the attribute types and the column field names in the meta file. Also, we create a tuple entry for every record and push it in the same meta file.

insertTuple(byte[] tuplePtr):
We get a byte array tuplePtr as an input for this function. We insert a tuple in the database. We get the TIDs and for every TID, we have an array of RIDs. Before inserting the records, we check whether the type of column data is String or Integer because we must have the length of the value while creating the byte array for the insertRecord() method. Iterating through the columns, we use the the insertRecord() method of the Heapfile class. We create a new TID object which contains the array filled with RIDs after we do the above method for every column. This new TID is then added to a page by conversion to a byte array.

getTuple(TID tid):
We take the TID as an input for this method. We use it to loop all the RIDs in its record array. or each RID we use the Heapfile object in the columnarFiles array to call getRecord() passing in that RID. We get a tuple in return and we copy this tuple into a byte array by incrementing the offset as we loop through it. We create a tuple object which contains the byte array that we add to it when we go through it in a loop.

getValue(TID tid, column):
We take the TID and the integer column value as an input to this method. We index into the columnarFiles array of heapfiles by using the the value of the integer column in the input parameter. We use the getRecord() on the heapfile using the rids arrays with column value to get a tuple in return. Then we have to check the type of the given column in order to create either a String or Int with the returned tuple value.

updateTuple(TID tid, Tuple newtuple):
We get a new tuple and a TID as input parameters for the method. We have to update the tuple at TID with the values inside the input newTuple object. Firstly, the newTuple is stored as a byte array. We then iterate the columns to and when we get a hit, we use the updateColumnofTuple() to update its contents. We then return a boolean based on the success of the operation

updateColumnofTuple(TID tid, Tuple newtuple, int column)
This method receives a TID, newtuple value, and a column index. We use the column index value to index into our “columns” array to access the specific Heapfile object for that column. We then call update record on that Heapfile with passing in the current RID (received by using the tid that is provided) and the newtuple value.

markTupleDeleted(TID tid)
This method receives a TID as an input. The purpose of this method is to mark the provided tuple as deleted. We fetch the tuple required using the TID from as well as all the RIDs it contains. We iterate to the appropriate tuple by checking its attribute type and mark it deleted which is an associated boolean value.

purgeAllDeletedTuples()
This function iterates over all the TIDs and checks whether the deleted boolean associated with the record. If it has been set to true, it invokes the deleteRecord() method of the Heapfile class to delete it. It performs the action all the RIDs present in the array of the TID class. Once it deletes the RIDs, it deletes the TID as well.

2.2 ColumnarMetaFile Class:

The ColumnarMetaFile is used to store metadata information for the data stored in the columns which are populated in the ColumnarFile. We store metadata such as the number of columns, their names, and size of tuples as the size of the tuple can vary according to the data which user creates in the database.

ColumnarMetaFile() - Constructor:
The constructor has been overridden in two different ways. The first constructor takes one parameter as input, which is the filename of String data type. In this we create a new Heapfile and then initiate an open scan to get all the tuples. The next constructor has an additional value stating the number of columns. In this we define the tuple size and create a byte object called metaInfo with the calculated tuple size.

getTuple()
This method is used to return the meta information of a Tuple and returns a tuple object. We set the meta values such as string_length, numColumns, and tupleLength. In this method we iterate over the number of columns to set the columnFieldNames array which is the meta info on the record entries, columnFileNames array which stores the columnar data, and the attributeTypes which states the type of data stored in the columns.

Index Query:
The purpose of creating the driver class Index is to create an index on the given column. This function takes four input parameters, namely, ColumnDB name, Columnar file name, column name and the type of index to be built. The two different types of indexes are BTree and BitMap. We pass these input as command line arguments to the main program. We first iterate over the file which has all the column files and get the column which is a heapfile of all the records by using the getColumnByName() method of the Columnarfile class.

BTree Index:
The creation of a BTree index depends on the attribute type of the column i.e. whether it's String or Integer. We start by doing an open scan on the column heap file by openScan() method of the Heapfile. A new BTree object is created with parameters including file name, key type, key length, and a delete fashion which is how you want to delete it. We use full delete fashion. Then we iterate over the tuples TID in the file by using the getNext() method of the Heapfile to receive the values of the columns being indexed. We do this using RID and we insert the values in our newly create BTree and then print the entire tree.

BitMap Index:
A BitMap index is based on the unique values present in a column. It's because a single Bitmap can only provide information regarding location of one particular file. So we need to create a whole array of BitMaps to create a good index on a column. We first check the attribute type of the column whether it's a String or an Integer. We then loop through the column in order to create a set of unique values. We then call on the getBitMapFile() method of the BitMapUtil class which is the constructor to create the bitmap and then we print the BitMap file in the console.

2.3 TID Class

TID() - Constructor:
We wanted to design the TID class such that it can be accessed by three different sets of parameters. Hence we override the constructor 3 times with the different types of parameters which can be passed. The first constructor has a variable "numRIDs" as its input which is of the integer type. That value is assigned to the instance variable in the class of the same name. The next constructor takes an additional integer parameter position which describes the position of the tuple ID. The last constructor takes three parameters along with the previous two. The third parameter is an array of record IDs.

copyTid(TID tid)
This method takes in a TID object and copies over the instance data to the instance object TID in the class.

equals(TID tid)
This method takes in a tuple ID as input and returns a boolean value which depends on the success of whether the tuples have the same instance variable values of numRIDs, positon, and recordIDs.

writeToByteArray(byte[] array, int offset)
This method takes two input parameters, one is a byte array and another is the integer offset. This method converts the instance data into byte array data type and uses the RID method writeToByteArray() for each RIDin the array containing recordIDs

2.4 Tuple Scan Class

This is a Class in the Columnar package. The purpose of implementing the Tuple Scan class is to get complete tuples which contains the data from simultaneously scanning all columns.

TupleScan() - Constructor
The constructor opens the scan on Heapfile objects from the file which contains all the columns and run and openScan on them. The openScan() function is a method of the Heapfile class.

closetuplescan():
This method iterates over the scans array and calls the closescan() method which is a function of the Heapfile.

getNext(TID tid):
This method takes the TID as an input parameter. This method iterates over the scans array which is present for each column and calls the getNext() method on each of the scan. After a tuple is fetched using the getNext() method, it's added to a bigger tuple which contains the data from all the scan run from the array. The method then returns a tuple which contains all the data corresponding to the requested TID from all the columns.

position(TID tid):
This method takes the TID as an input parameter. This method iterates over all the RIDs in the given TID's array. For each of these RIDs, we use the position() method to position the scan cursor to be at the required tid. It returns a boolean value based on the success of whether the position of the tuple has set or not.

2.5 Bitmap Implementation

The given phase uses bitmap indexing as one of the indexing techniques. The module includes header file and its page structure which are the components related to the bitmap file.
The following syntax invokes the creation of the Bitmap file and its corresponding header page:
Index COLUMNARDB COLUMNARDBFILE COLUMNNAME BITMAP

BitMapFile:
The BitMap file extends heapfile and it works similar to Minibase's header page implementation for a BTreeFile.
It is the main container file for bitmap. It contains columnarfile, valueclass, bitmapheaderpage. This file creates bitmap on the attribute value that is been set. Before creating Bitmap Index on any columnar file, we need to determine the datatype as well as the number of columns that are to be indexed. When the input commands for it is entered by the user on console, The ColumnarIndexTest.java file invokes the method createBitMapIndex() of the columnar
File and a new heap file is created(BITMAP_COLUMNNAME_COLUMNVALUE).

2.6 BitMapHeaderPage:

This extends HFPage, it contains list of BMPage where the actual bitmap data is stored, constructor initializes the pages at the bitmap file creation. Each header file forms a metadata on file and has a fixed set of property like root_id, key_type, maxKeySize, deleteFashion.
For every heap file that is created, the system sets some header information, calls BitMapHeaderPage.java constructor and and the init() method is invoked as well.

init():
When this method of BitMapFile.java class is invoked, the necessary values and the heap file containing thedata of the column which is to be indexed is passed and createBitMap() method is called after all state variables are set.

While scanning the heap file, we compare the byte array of every value by fetching it's RID.
The two methods from BitMapFile.java class -> get_1() and get_0() sets the byte array to 1 or 0 for match and mismatch respectively and is inserted accordingly in the newly created bitmap file till all the values in the column are matched with the given equality criterion value of the query.
After creation file, it is printed on the console for logging and debugging purpose and the printing is done by scanning the Bitmap Index file, getting each record using heap file's getRecord method. The position of each tuple is also printed along with the bit value.

2.7 ColumnarDBTest Class:

This class belongs to the tests package. It takes the command-line argument of the driver class to be executed. It breaks down the command received and invokes the CommandUtils class to execute the particular to be executed. It sends the necessary test to be run along with the arguments passed by the user.

2.8 CommandUtils Class:

This class also belongs to the tests package. It contains functions with all the test cases which we perform on the database.It contains the following classes:
BatchInsert()
Index()
Query() and
Delete_Query()
The detailed explanation is provided in each of the individual sections.

2.9 BatchInsert Implementation:

BatchInsert inserts raw data from external data file into database. The Batch insert program takes following as input parameters:

1. Data file name with path 2. ColumnDB name 3. Columnar File Name 4. Number of columns

Example command:
Batchinsert C:\Users\PAVAN1\Documents\GitHub\columnar\src\tests\smalltest.txt testCol Colcr 4

Program first checks for number of arguments if they are invalid it throws invalid argument error. After that program reads the first line of the input file and gets the schema and column name and data type information. Program then creates columnar file after getting data. At the end of the insertion program uses PCounter class to display total number of disk page reads and writes.

2.10 Query.java

This is the driver class for query
Syntax :
Query COLUMNDBNAME COLUMNARFILENAME [TARGETCOLUMNNAMES] VALUECONSTRAINT NUMBUF ACCESSTYPE

This program access the database and print matching results, It also counts the number of read and write disk operations.
The VALUECONSTRAINT specifies the matching condition in the format
ColumnName:Operator:Value eg: Salary:=:5000
The TARGETCOLUMNNAMES specifies the attributes to select from the database and display in the result.
NUMBUF is the maximum number of buffer pages that the query uses.
ACCESSTYPE specifies the type of Access like FileScan, Column scan, B Tree or Bitmap.

parseValue(String tempValue):

This function takes the VALUECONSTRAINT as a string input and parse it to fetch Column Name, Operator and Value.

getOutputTuples(int position)

This is used in Access type Column scan, B Tree and Bitmap.
This function takes the position of the record in the ValueConstraint and gives all the records in the target columns at the same position.

getFileName(Tuple t)

This is used in Access type FileScan.
This function takes the Tuple of the matching condition record and gives all the records in the target columns at the same position.

main()

We read all the arguments from the command line , create a db from the the given name , a columnar file, and open scan as per the access type and search the record as per matching condition. After getting the position of record we fetch records from other target columns at the same position.

2.11 Delete Query:

Implementation of delete query program is similar to Query program but it deletes the selected tuples.
Example command: delete_query testCol Colcr A B C D C<4 1000 BITMAP false

Delete query also has a last parameter as boolean which is tells it if tuples should be purged from the database or not.

3. Interface specifications:

Internal Memory was used for the data storage.

The java core program provides a command line interface to enter the program name and command line parameters

Internal Memory was used for the dataset storage.

Command Prompt is used to execute entered commands and run batch files.

Minibase for Java: Modification of minibase for java is used for implementation of columnar.

The interface used to interact with the columnar database has five menu options:
1. Insert
2. Index
3. Query
4. Delete
5. Exit

4. System requirements/installation and execution instructions:

a. Software requirements:
- i. JDK 1.8
- ii. JVM
- iii. Code base

b. Hardware Minimum Requirements:
- i. 2 GB of RAM memory
- ii. 2 CPU cores
- iii. 1 GB of storage space
- iv. 32 bit processor

c. Installation and Execution Instructions:

Initially, download and extract JDK. After making all the changes,
Un-tar the compressed code using

$ tar -xvzf cse510-dbmsi- project.tar.gz

2. Get into the tests directory

$ cd cse510-dbmsi- project/javaminibase/src/tests/

3. Change the following properties in the Make file under tests

JDKPATH: Set the path of the JDK 1.8 installation's home directory

4. Run the make command

$ make columnartest

5. The menu will pop-up and the interface will ask the user to choose between the four different operations

d. Execution Instructions:

To insert tuples into the existing table:

Batchinsert Datafilename Columndbname Columnarfilename Numcolumns

Where Datafilename, Columndbname, And Columnarfilename Are Strings While Numcolumns Is Integer

To implement a program index:

Index Columndbname Columnarfilename Columnname Indextype

Where Columnname, Columndbname, Columnarfilename, And Indextype Are All Strings.Indextype Is Either Btree Or Bitmap.

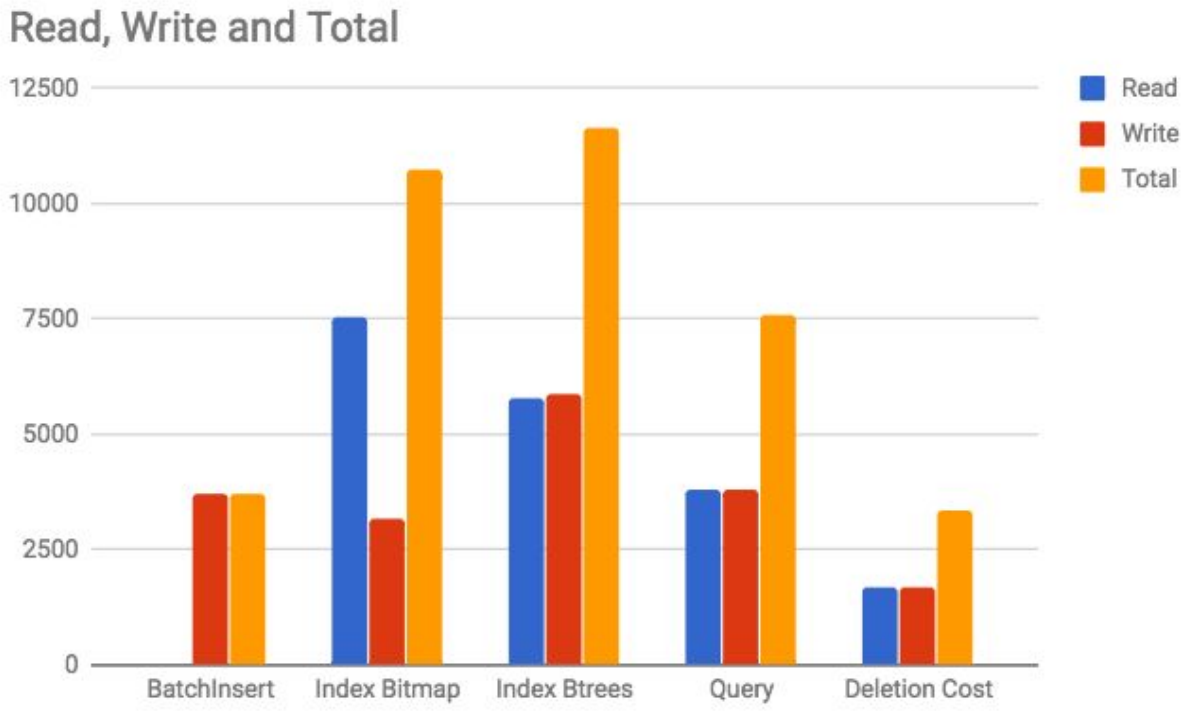
To implement a program query:

Query Columndbname Columnarfilename [Targetcolumnnames] Valueconstraint Numbuf Accesstype

5. Cost Analysis

We will now tabulate and plot the the cost of reads and writes we get after running each if the tests on our database.

Test Driver	Batch Insert	Index		Query	Delete Query
		BitMap	BTree		
Page Reads	14	7548	5769	3792	1683
Page Writes	3722	3170	5890	3787	1682
Total	3736	10718	11659	7579	3365



6. Related work

SURVEY PAPER	AUTHORS	DESCRIPTION
The Design and Implementation of Modern Column-Oriented Database Systems	Daniel Abadi Peter Boncz Stavros Harizopoulos Stratos Idreos Samuel Madden	The authors described a number of architectural innovations like vectorized processing, late materialization, compression and database cracking which made modern column-stores like MonetDB, VectorWise and C-Store able to provide very good performance on analytical workload. They compared and concluded that . MonetDB and also expected that column-oriented ideas would find their way into other data processing systems, such as Hadoop/MapReduce.
The Impact of Columnar	Hasso Plattner	The paper proposed a read-only replication of the transactional schema where the scale-out

In-Memory Databases on Enterprise Systems		is performed by shipping the redo log of the master node to replications that replay transactions in batches to move from one consistent state to the next and the data permanently resident in main memory. The powerful master node handles transactions and OLXP workload with strong transactional constraints. The authors believe that the proposed might replace traditional row-based databases
Column-Stores vs. Row-Stores: How Different Are They Really?	Daniel J. Abadi Samuel R. Madden Nabil Hachem	The presented paper the comparison in the performance of C-Store to several variants of a commercial row-store system on the data warehousing benchmark, SSBM was made and it was shown that the attempts to emulate the physical layout of a column-store in a row-store via techniques like vertical partitioning and index-only plans do not yield good performance. It was concluded that a successful column-oriented simulation will require some important system improvements, such as virtual record-ids, reduced tuple overhead, fast merge joins of sorted data, run-length encoding across multiple tuples, and some column-oriented query execution techniques like operating directly on compressed data, block processing, invisible joins, and late materialization

7. Conclusion & Future Scope:

Conclusion:
In the presented phase we observed the cost difference between indexing strategies and scans and also understood the inner workings of a columnar database. We observed that the bitmap index and columnar scan performs better than a B+ tree and a file scan.
We also observed that a column oriented database performs better than a row oriented database for the retrieval of data but when the number of columns projected increase, it starts getting closer to the way row-store works and hence the performance decrease.

Future Scope:
Compression algorithms can be used on columns to reduce the scan timing as well as space

required to store the data.
The data could be made persistent and the scope should be increased beyond the session.
More conditions can be added to value constraints.

8. Bibliography:

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The Design and Implementation of Modern Column-Oriented Database Systems," Foundations and Trends in Databases, vol. 5, no. 3, pp. 197-280, 2013.

[2] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Proceedings of the ACM SIGMOD Conference on Management of Data, pages 671–682, 2006

[3] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. Proceedings of the Very Large Data Bases Endowment (PVLDB), 5(10):1064–1075, 2012

[4] Plattner, Hasso. "The Impact of Columnar In-Memory Databases on Enterprise Systems." *PVLDB* 7 (2014): 1722-1729.

[5] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? ACM, 2008.

[6] M. Faust, D. Schwalb, J. Krüger, and H. Plattner. Fast lookups for in-memory column stores: Group-key indices, lookup and maintenance. In ADMS@VLDB, 2012.

[7] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. VLDB, 2005.

[8] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 967–980. ACM, 2008.

[9]M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In VLDB, pages 553–564, 2005.

[10] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

[11] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In ICDE, pages 466–475, 2007.

9. Appendix:

Columar and ColumnarTest was done by Ankit.
Bitmap was done by Prashant, Saumya
Driver classes were done by Shivam, Pawan ,Pranjal, Mahima and Ankit.
Heapfile Classes by Mahima and Ankit..
Scans were done by Mahima, Shivam, Pranjal, and Saumya
Report was done by Saumya, Pranjal, Mahima, Shivam, Prashant, Pavan and Ankit.