CSE – 510 PROJECT PHASE II

IMPLEMENTATION OF COLUMN STORES ON ROW STORES AND PERFORMANCE ANALYSIS

(GROUP – 3)

Archana Ramanathan Seshakrishnan (1211112666)

Soumya Adhya (1213191315)

Aditya Chayapathy (1213050538)

Jithin Perumpral John (1213175858)

Kurra Dixith Kumar (1211262790)

Ejaz Saifudeen (1213262763)

ABSTRACT

This phase of the project focuses on the implementation and analysis of column stores on top of row stores. Though row stores perform very well on transactional workload, they fail to show the same level of efficiency on OLAP workload. Columnar data stores leverage the idea of storing data pertaining to a single attribute together. With this, OLAP operations such as aggregations are simplified as they involve accessing only those attributes of interest. In this report, we experimentally analyze the read and write performance of different indexing alternatives for batch insertions and for different query types.

Keywords: row-store, column-store, attribute, equality search, range search, relational DBMS, bitmap index, b tree index

I. INTRODUCTION

- **Goal Description:**

Javaminibase is relational database management system (Relational DBMS) intended for educational use. The implementation is the traditional rows stores. This implementation is used as the base on which we build a column store. Though a significant performance improvement is observed in certain queries like select (a particular column) with equality and range searches, this model does not seem to perform well in case of full scan. This also gives insight on the applications where we can benefit from column stores like analytics on an attribute. The actual implementation of column stores is different from what has been implemented in this phase. This is mainly because in column store there is a significant design changes at query processor and storage level.

- **Assumptions:**

1. The name of any columnar file doesn't exceed 100 characters.
2. The value of any attribute in the columnar file used for bitmap indexing doesn't exceed 400 bytes.
3. This version of columnar database supports 2 datatypes namely, Integers and Strings.
4. The number of pages allocated to a database is fixed with 10000 pages.
5. The size of a page is fixed with 1024 bytes.
6. When a B+ Tree index is created on a column having integer values, we assume it's values are positive integers.
7. The order of insertion is not relevant and **hence it is not preserved.
8. When the IndexType is a bitmap, the target columns of type String or Integer are supported and any other type would throw a valid exception
9. When the IndexType is a bitmap, only equality searches are supported.
10. Whenever we perform an index scan using btree, the inequality operation is not supported.
11. Number of columns given in the command line is equal to that in the sample file provided.
12. In BTree, we use the iterators provided by Minibase.
13. It is assumed that everything is inserted and retrieved from a single DB.
14. Limitation: Deletion using Btree Index were not working as expected.

- **Terminology:**

Database: A database is an organized collection of data [1].

Database Management System: A database-management system (DBMS) is a computer-software application that interacts with end-users, other applications, and the database itself to capture and analyze data [3].

Data Model: A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of the real-world entities [2].

Data Schema: The database schema of a database system is its structure described in a formal language supported by the database management system (DBMS) [3].

Relational Algebra: Relational algebra is a family of algebras with a well-founded semantics used for modelling the data stored in relational databases and defining queries on it [4].

Relational Calculus: Relational calculus consists of two calculi, the tuple relational calculus and the domain relational calculus, that are part of the relational model for databases and provide a declarative way to specify database queries [5].

SQL: SQL (Structured Query Language) is a domain-specific language used in

programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS) [6].

Buffer Manager: The buffer manager reads disk pages into a main memory page as needed. The collection of main memory pages (called frames) used by the buffer manager for this purpose is called the buffer pool. This is just an array of Page objects. The buffer manager is used by (the code for) access methods, heap files, and relational operators to read/write/allocate/de-allocate pages [7].

Parser: A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree [8].

Optimizer: A query optimizer is a critical database management system (DBMS) component that analyzes Structured Query Language (SQL) queries and determines efficient execution mechanisms. A query optimizer generates one or more query plans for each query, each of which may be a mechanism used to run a query [9].

Heap Files: A heap file is an unordered set of records [10].

Disk Space Manager: The disk space manager (DSM) is the component of Minibase that takes care of the allocation and deallocation of pages within a database. It also performs reads and writes of pages to and from disk and provides a logical file layer within the context of a database management system [11].

B+ Trees: A B+ tree is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children [12].

Column Oriented Database: A column-oriented DBMS (or columnar database management system) is a database management system (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. Query performance is often increased as a result, particularly in very large data sets [13].

Index: An index or database index is a data structure which is used to quickly locate and access the data in a database table. Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed. [14]

BitMap: A bitmap is an array data structure that compactly stores bits. [14]

Bitmap Index: A bitmap index is a special kind of database index that uses bitmaps. Bitmap indexes answer queries by performing bitwise logical operations on these bitmaps. [16]

II. IMPLEMENTATION

- **Value class:**

Purpose:

To abstract data types.

Implementation:

Value class is a generic abstract class that is used to give a generic value for all the types in the table namely String or Integer. There are three other classes namely ValueInt, ValueString which extends the Value

class to return Integer or String values respectively. Value Factory class ties up all this together in making the parent reference store the corresponding child instance.

- **TID class:**

Purpose:

Tuple ID (TID) and Record ID (RID) don't differ much in row stores. But they differ significantly with respect to column stores. In column stores the row is segmented into columns and stored separately in each file. TIDs are basically used to store a collection of RIDs as a binder for the reconstruction of the original row.

Implementation:

TID class has three attributes namely numRIDs, position and an array of RIDs named as recordIDs. "numRIDs" attribute is used to indicate the number of RIDs, the position to indicate the relative position in the original table and the array of RIDs for reconstruction. The constructor is overloaded thrice each initializing one, two or all the member variables.

Methods:

copyTID – takes a TID argument and returns nothing. It copies all the values of the instance variables of the argument to that of the called object's members.

Equals – takes a TID argument and returns a Boolean value it checks the equality of all the member variables between the called object and the argument.

writeToByteArray – takes a byte array and offset as arguments and returns nothing. By convention the first 4 bytes of the offset is for the number of RIDs, the next four is for the position. The rest data is the array of RIDs. Since RIDs contain Page number (integer member variable) and slot number (int).

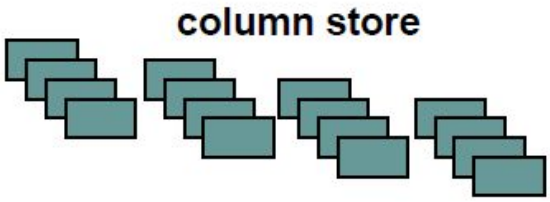
setPosition – takes integer argument for position and returns nothing. This function is used to set the position.

setRID – takes RID argument and returns nothing. This function is used to copy the values from the argument to member variable.

- **Columnarfile class:**

Purpose:

This class is used to design the Columnar file as a combination of heap files, number of columns, attribute sizes, asizes (to accommodate the extra 2 spaces for strings), name of the file scan, first record, RID from the header file, HashMap to. This class defines the different columns and store them to different heap files.



Implementation:

The following conventions are followed in the upcoming methods.

1. Attributes of a tuple are split across an equal number of heapfiles, ie each heapfile represents a column. The tuple structure is not used in these heapfiles, data is just stored as byte arrays without metadata. This is done to conserve space. The attribute_type and attribute_size is instead stored in the columnar header. Other than the column heap files all other heapfiles contain records in the minibase tuple format.

2. The name of the columnar file is stored in member variable "fname".

The name of the header file is fname+".hdr".

3. An optional heapfile can be part of the columnar file. "columnar_name.idx" contains the names of all indexes(Btree or Bitmap) created on columnar attributes.

4. Another optional heapfile is the "columnar_name.del" file which is used to keep track of all tuples that have been marked for deletion but have not been purged yet.

5. The name of the heapfiles for each of the columns is fname + column_number. Hence the number of heapfile objects is number of columns + 1 (+1 for .idx file)(+1 for .del file).

6. The organization of data in the header heap file is as follows: first field contains the number of columns, followed by attribute1 type, attribute1 size, attribute1 name….attribute n type, attribute n size, attribute n name.

7. The string sizes are taken as input from the user.

Methods:

Columnarfile(java.lang.String name) – takes the columnar file name as argument. This constructor is for fetching the existing file. It also populates values to the member variables as described above. The array asizes is modified depending on the attribute type. The exceptions here are properly handled in the event of missing files.

Columnarfile(java.lang.String name, int numcols, AttrType[] types, short[] attrSizes, String[] colnames) – takes columnar file name, attribute sizes, attribute types and the number of columns as arguments.

void **deleteColumnarFile**()- This method deletes all the relevant columnar files which are created. The files for all the columns are deleted.

TID **insertTuple**(byte[] tuplePtr)- This method takes in a row tuple as a byte array. Individual attribute values are extracted and inserted into the relevant heap file and TID is constructed. This method also checks for existing indexes and inserts the key/position in the relevant index

Tuple **getTuple**(TID tid)- This method takes one TID as an argument, and the tuple is returned from the Columnar files. The records from all columnar files are read using the getRecord function and copied to a byterarray, which will be merged together as a tuple using the tupleInit function and is returned.

ValueClass **getValue**(TID tid, column)- This method takes a tid and column number as input, and returns the actual value of the column at this tid. Method heapfile.getRecord() is called to retrieve the particular column of the tuple and then Valuefactory.getValue class is called to return the actual value of the function.

int **getTupleCnt**()- This function return the number of records in the columnar file. It calls the heapfile.reccnt to return the number of records.

TupleScan **openTupleScan**()- This is an overloaded functions, which take either no input or columns as inputs. They internally call tupleScan(this), which instantiates scans on all the columns, or tupleScan(this, columns), which instantiates scans on the provided columns.

Scan **openColumnScan**(int columnNo)- This function takes the input as a columnNo and

instantiates a scan on the particular column heap file.

boolean **updateTuple**(TID tid, Tuple newtuple)- This method takes a tid an new tuple as arguments and updates the records in each columnar file. It internally calls heapfile.updateRecord(), for each columnar file.

boolean **updateColumnofTuple**(TID tid, Tuple newtuple, int column)- This method takes TID, newTuple, and a column number as input and updates the record for a particular column at the tid. It internally calls heapfile.updateRecord() for the particular input columnar file.

boolean **createBTreeIndex**(int column)- This function takes a column number as input and creates an unclustered BTree index on this columnar file. It creates the BTree index using BTreeFile() constructor and then the corresponding columnar file is opened and the records from this file is inserted to the btree file using btree.insert() function.

boolean **createBitMapIndex**(int columnNo, valueClass value- This function takes a column number and a value as input, and a new bitmap file is created using BitMapFile() as constructor, by passing this columnNo and value. Then, the corresponding columnar file is read and a 1 is inserted, using the function bitmapfile.insert(), if this value is found at this position, or 0 is inserted, using the function, bitmapfile.delete(), if the value is not found.

boolean **createAllBitMapIndexForColumn**(int columnNo)- This function is used to create bitmap indexes for all unique values in column columnNo.

boolean **MarkTupleDeleted**(int position)- This method is used to mark a tuple to be deleted. The position of the tuple is taken as an input. The tuples which are marked to be deleted are kept in a new heap file. We store only the position of the record which needs to be deleted in the new heap file. Then, we verify whether any indexes are created respective to this record (bitmap) or column(btree).

We check whether a Btree index is created on any of the columns present. If a Btree index is created on the columnar file, then the Key value and the RID of the element is passed for this key to be deleted from the Btree index.

Then, a check is performed to verify whether a Bitmap index is created for any of the value in the tuple. If a bitmap index is created, then the position of the tuple is passed to the bitmap delete function, which will set this value to 0.

boolean **PurgeDelete**()-This method is used to perform deletion of the records, which are marked for deletion, from Columnar files. This function opens the heapfile where the records are marked for deletion, and it iteratively delete the record from each columnar files at that position. The btree indexes were deleted in the MarkTupleDeleted function.

After the columnar record is deleted, we need to delete the corresponding entry in the bitmap index file.

String **getBTName**(int columnNo)- This function returns the btree file name.

String **getBMName**(int columnNo, ValueClass value)- This function returns the bitmap index file name.

String **getDeletedFileName**()- This method returns the file name of the marked records for deletion.

- **TupleScan class:**

Purpose

This class is used to scan through the tuples in a heap file.

Implementation

Methods

TupleScan(Columnarfile f)- This function initiates a scan on all the columns.

TupleScan(Columnarfile f,short[] columns))- This function initiates a scan only the column which is provided. This constructor truly takes advantage of the columnar organization by scanning only the required column file.

Tuple **getNext**(TID tid)- This function takes a tid argument and returns a tuple as output.

BITMAP FILE DESIGN

Fig1 describes the structure of a bitmap index implementation. A bitmap file is composed of a header page (BitMapHeaderPage) and a list of bitmap pages (BMPage) linked to each other in the form of a doubly linked list. The header page is the first page in the linked list followed by one or more bitmap pages.

The header page maintains all the metadata needed for the bitmap index. The metadata information stored in the header file are:

1. The page id of the first BMPage in the doubly linked list.

2. Name of the columnar file.

3. Column number of the columnar file on which the bitmap index is created.

4. Attribute type of the columnar file on which the bitmap index is created.

5. Attribute value of the column on which the bitmap index is created.

A bitmap page is used to store the bitmap for the index. BMPage is divided into 2 sections:

1. Header: This section is used to store the metadata with respect to the bitmap page.

2. Data: This section is used to store the bitmap data.

The metadata maintained in the header section are:

1. Counter: It is a short integer that indicates the number of bytes in the data section that are a part of the bitmap.

2. Available Space: It is a short integer that indicates the available space (in bytes) present in the bitmap page.

3. Previous Page: It keeps track of the page id of the previous page in the doubly linked list.

4. Current Page: It keeps track of the page id of the current page.

Next Page: It keeps track of the page id of the next page in the doubly linked list.

- **BM class :**

Purpose:

This is a bitmap specific utility class**.**

Implementation:

Methods:
BM() - This is a default constructor.

printBitMap(BitMapHeaderPage header) - This method is used to pretty print the contents of the bitmap file. It can be used for debugging purposes.

- **BitMapHeaderPage**

Purpose:

The metadata with respect to the bitmap index are stored in these pages.

Implementation:

It extends the "HFPage" class and adds the metadata as described above.

Methods:

BitMapHeaderPage(PageId pageno) - It takes the page ID of an existing BitMapHeaderPage as input and pins the page to the buffer.

BitMapHeaderPage(Page page) - It is a constructor that takes a page as input and makes this BitMapHeaderPage point to the given page.

BitMapHeaderPage() - It is the default constructor. It initializes the page with the necessary metadata with default values.

void **dumpHeaderPage**() - This method prints all the metadata information present in the BitMapHeaderPage. It can be used as a debugging utility to view the contents of the header page.

void **setColumnNumber** (int columnNumber) - This is a setter method for the "Column Number" field.

void **setAttrType**(AttrType attrType) - This is a setter method for the "Attribute Type" field.

void **setColumnarFileName** (String columnnarFileName) - This is a setter method for the "Columnar File Name" field.

void **setValue**(String value) - This is a setter method for the "Value" field.

Integer **getColumnNumber**() - This is a getter method for the "Column Number" field.

AttrType **getAttrType**() - This is a getter method for the "Attribute Type" field.

String **getColumnarFileName**() - This is a getter method for the "Columnar File Name" field.

String **getValue**() - This is a getter method for the "Value" field.

PageId **getPageId**() - This is a getter method for the "PageId" field.

void **setPageId**(PageId pageno) - This is a setter method for the "PageId" field.

PageId **get_rootId**() - This is a getter method for the "First BMPage Page ID" field.

void **set_rootId**(PageId rootID) - This is a setter method for the "First BMPage Page ID" field.

- **BitMapFile:**

Purpose:

This class is used to perform all the operations on the bitmap index.

Implementation:

BitMapFile(String filename, Columnarfile columnfile,int ColumnNo, valueClass value) - This constructor is used to create a new bitmap file. It involves creating a new BitMapHeaderPage and setting it's metadata parameters accordingly.

BitMapFile(String filename) - This constructor is used to open an already existing bitmap.

Method:
void **close**() - This method unpins the header page from the buffer and closes the file.

void **destroyBitMapFile**() - This method opens and frees all the pages associated with the BitMapFile.

BitMapHeaderPage **getHeaderPage**() - The is a getter method to obtain the header page of the bitmap file.

boolean **Delete**(int position) - This method is used to set the value of bitmap to '0' at the position specified as input. This step involves checking if the position specified can be accommodated in the given set of bitmap pages (BMPage) and dynamically adding new BMPage if necessary.

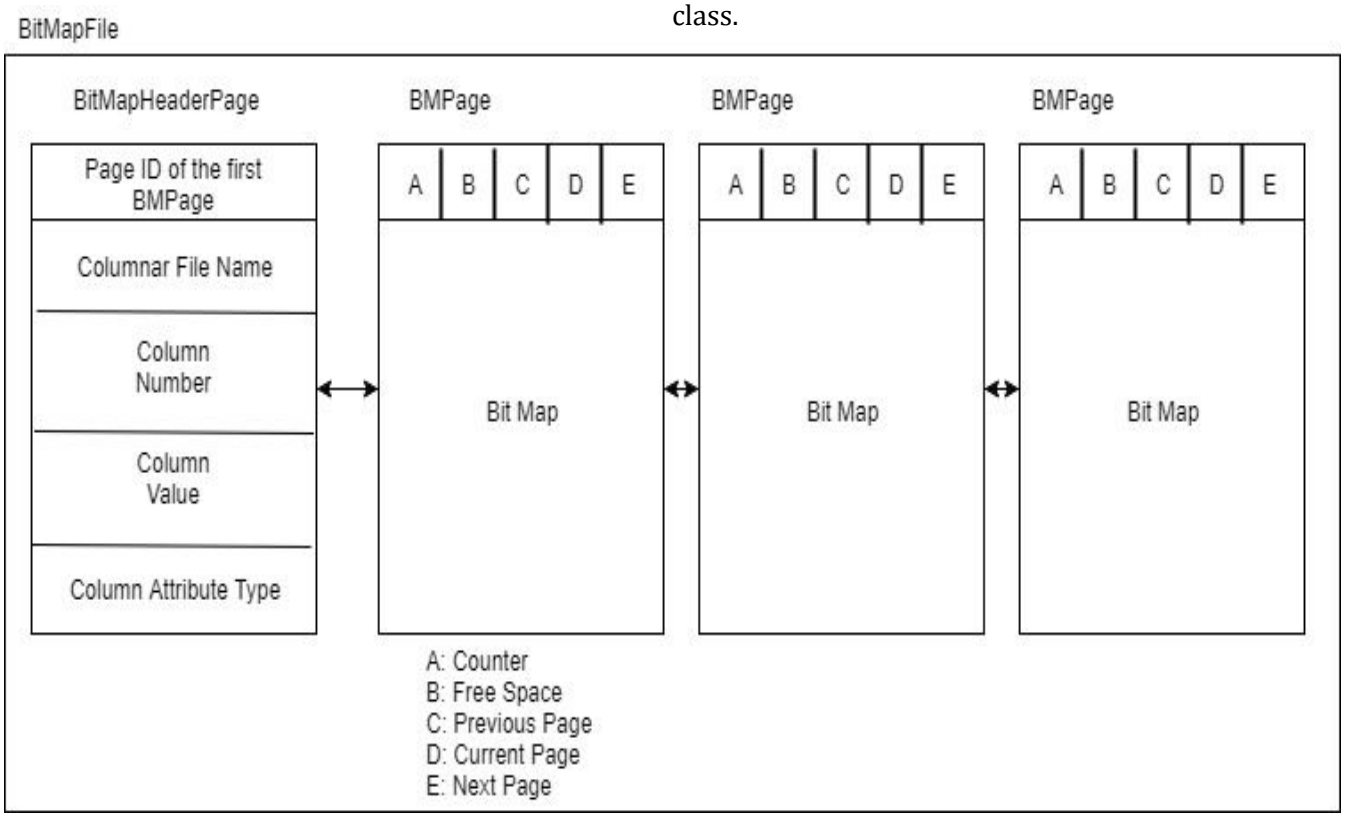


Fig 1: BM File Implementation

boolean **Insert**(int position) - This method is used to set the value of bitmap to '1' at the position specified as input. This step involves checking if the position specified can be accommodated in the given set of bitmap pages (BMPage) and dynamically adding new BMPage if necessary.

- **BMPage class:**

Purpose:

The data with respect to the bitmap index are stored in these pages.

Implementation:

It extends the "Page" class and partitions the data (byte array) into 2 sections as described above**.**

Methods:

BMPage() - It is a default constructor for the class.

BMPage(Page page) - It is a constructor that takes a page as input and makes this BMpage point to the given page.

int **available_space**() - It returns the amount of available space on the page. To calculate this, we fetch the "Available Space" section in the header portion of the BMPage.

void **dumpPage**() - This method prints all the relevant information present in the BMPage.

It can be used as a debugging utility to view the contents present in both sections of the BMPage.

boolean **empty**() - This method is used to determine if the page is empty or not. To calculate this, we fetch the "Available Space" section in the header portion of the BMPage. Next, we compare this value against the maximum size of the page to determine if the page is empty.

void **init**(PageId pageNo, Page apage) - This method initializes the header and the data section of the BMPage. The following are default values set for the headers:

1. Available Space - Size of page - size of header section
2. Counter - 0
3. Current Page - Page id of the current page
4. Previous Page - INVALID
5. Next Page - INVALID

The remaining bytes are initialized with the value '0'.

void **openBMpage**(Page apage) - This method assumes the input is an existing BMPage. This method then copies the contents (byte array) of the input BMPage to the current BMPage.

PageId **getCurPage**() - This is a getter method for the "Current Page" field.

PageId **getNextPage**() - This is a getter method for the "Next Page" field.

PageId **getPrevPage**() - This is a getter method for the "Previous Page" field.

void **setCurPage**(PageId pageNo) - This is a setter method for the "Previous Page" field.

void **setNextPage**(PageId pageNo) - This is a setter method for the "Previous Page" field.

void **setPrevPage**(PageId pageNo) - This is a setter method for the "Previous Page" field.

byte[] **getBMpageArray**() - This method returns the data section of the BMPage.

void **writeBMPageArray**(byte[]) - This method sets the data section of the BMPage.

void **updateCounter**(value) - This method updates "Counter" and the "Available Field" fields.

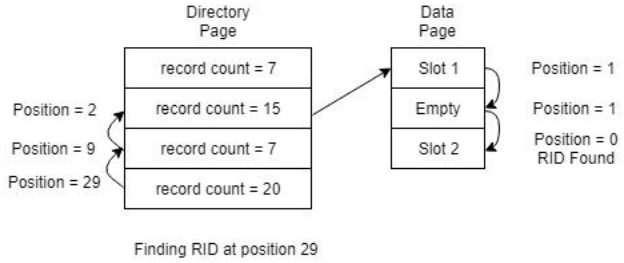
Integer **getCounter**() - This is a setter method for the "Counter" field**.**

- **HeapFile class (modification):**

Method:

PageId **setCurPage_forGivenPosition(**int position)- This function takes position as an input and set the value of current page to the page that contains the entry at this position, and returns the pageID.

RID recordAtPosition(int position) - This function is used to iterate through the directory of a heap file to locate the data page containing the RID of interest, then the slots of the datapage are iterated to get the relevant slot. The diagram gives a better visual representation of this process



Finding RID at position 29

int position(RID r) - Does the process of recordAtPosition in reverse.

- **ColumnDB class (modification):**

Purpose:

This class is similar to DB class of minibase. It takes responsibility of reading, writing, and allocating the pages. It maintains

a directory which keeps track of allocated pages and memory.

Implementation:

The DB class is extended to columnarDB, and the below two functions are overridden:
void **read_page**(PageId pageno, Page apage)- In this method, the pcounter's read value is incremented.

void **write_page**(PageId pageno, Page apage)- In this method, the pcounter's write value is incremented.

- **iterator.ColumnarFileScan class:**

Purpose:

It's used to scan the file with a predicate. We call this function to return tuples which satisfy the condition.

Implementation:

This class is extended from the iterator class, and it reads the columnar file to be scanned as input along with the conditions. The following methods are implemented:

Methods:
ColumnarFileScan(java.lang.String file_name, AttrType[] in1, short[] s1_sizes, short len_in1, int n_out_flds, FldSpec[]proj_list, CondExpr [] outFilte, short[] in_cols)- This constructor takes input the columnar file name to be opened, array showing what the attributes of the input fields are, shows the length of the string fields, number of attributes in the input tuple, number of fields in the out tuple, shows what input fields go where in the output tuple, select expressions.
It checks whether the columnar file and a scan on it could be opened.

FldSpec[] **show() -** This file shows what input goes where in the output tuple.

Tuple **get_next()**- This function returns the next record which matches the predicate. The predicate condition is checked using the function 'PredEval.Eval', and tuple will be returned using the function: Projection.Project.

bool **delete_next()**- This function marks the next record which matches the predicate for deletion. The predicate condition is checked using the function 'PredEval.Eval', and a boolean will be returned denoting the success of the delete operation.

- **index.ColumnIndexScan class:**

Purpose:

Column store is different from the row stores in terms of the storage structure, where each column is stored separate table. This store structure can be leveraged to scan the required columns rather than scanning the entire row

Index Scan (BitMap):

Bitmap index files are created on the individual unique value Class for the particular column in column store as explained above. These bitmap files can be used as an index to scan the col store based on the value constraints

Implementation:

Columnar Index Scan, currently supports only equality search, can be easily extended to other scans like inequality and range search.

The bitmap file needed to scan is inferred from the valueConstraint, given a value constraint on columarDB called College, ColumnarFile called students, Value constraint A Eq 5, then the bitmap file with name College-students-A-5 is opened. The respective bitmaps are scanned sequentially from the BMPages of the opened bitmap file

The respective coulmnarfile is opened

using the relname, which is the concatenation of the columarDB and coulmnarFile, indexType is set to bitmapIndex, an Array of Heapfiles are intialized respective targeted columns of the ColumarFile are opened, targetAttributesTypes which is an array of attribute types which needs to be projected and targetAttributeSizes is initialized with the the individual sizes of the projections.

The columnar index scan is an iterator and yields a tuple on each of the get_next() call. The tuple header is set based on targetAttributeTypes, targetAttributeSizes. The bitmap file is iterated on each get next call and for every value of one, corresponding entries from each of the targeted columns are extracted and set into the tuple

Once the all the entries in the current BMPage are read which is being tracked using the BMpage's counter and scan counter, the next BM page is opened and the bits are read for scanning and tuples are read from those respective positions in the targeted heap files

- **Pcounter class:**

Purpose:

This class is used to implement the number of reads and writes.

Implementation:

This class is implemented using two variables rcounter and wcounter, and the below two methods. These functions are called from columnDB.readPage and and columnWritePage:

Methods:

static void **readIncrement**(): Increment the number of reads by incrementing the static variable rcounter.

static void **writeIncrement**(): Increment the number of writes by incrementing the static variable rcounter.

III. INTERFACE SPECIFICATION

Following the requirements specified in the specification document separate java classes were implemented for different testing scenarios.

1. **Batch Insert** :

Parameters : 1. Name of data file to be read 2. Name of the database 3. Name of the columnar file 4. Number of Columns

Test Case File Name : BatchInsert.java

Interface Implementation :

Query: batchinsert DATAFILENAME COLUMNDBNAME COLUMNARFILENAME NUMCOLUMNS

eg: java tests/BatchInsert sampledata.txt testColumnDB students 4

In this test case, the text file which was passed as input parameter was opened and parsed. Column names and their corresponding size was retrieved by traversing the first line. column values were retrieved by parsing the following lines and split using 'tab' character.

For each traversed line a single tuple was populated by the values that we retrieved after parsing the line.

Finally that tuple was inserted as a byte array.

As an outcome, we specified whether the test case executed successfully or not and if the insertion is successful this query prints the number of writes and reads.

2. **Create Index** :

Parameter : 1. Name of database 2. Name of columnar file 3. column name on which

indexing should be created 4. Type of index that will be created (BitMap or Btree).

Test Case File Name : IndexCreateTest

Interface Implementation : Method for creating b-tree and bitmap index on a particular column is implemented in columnar file. So in this interface columnar file is opened and corresponding create index is called depending on the type of index that is passed as a parameter. Columnname is also passed as a parameter to the columnar method for creating index.

As output it is specified whether the test case ran successfully and display the total number of reads and writes.

eg: java tests.IndexCreateTest testColumnDB students C BITMAP

3. **Select-Query** :

Parameters : 1. Name of the database 2. Name of the columnar file name 3. Targeted column names separated by commas 4. value constraint which consists of a column name, comparison operator and comparison value. 5. Number of buffers 6. Access type with possible values BitMap, Btree, FileScan and ColumnScan.

Test Case File Name : SelectQuery.java

Interface Implementation :

In this project **columnarfilescan** class is implemented to handle file scan, **columnarcolumnscan** class is implemented for column scan and in columnar file **columnarindexscan** method is added to handle scanning based on btree or bitmap index.

According to access type either columnIndexScan or columnarFileScan or columnarcolumnscan is invoked.

For BTree or BitMap columnIndexScan with specified access type is called and for FileScan ColumnarFileScan and for ColumnScan ColumnarColumnScan object is initialized.

From this methods we get a scan object which are traversed to get the tuples which satisfies the querying condition.

As an output we are displaying the output tuples and number of writes / reads made by the query.

eg: java tests.Select_query "SELECT testColumnDB students A,B {C > 5} 100 FILESCAN"

4. **Delete-Query :**

Parameters : Parameters passed in delete query is same as query with one extra parameter specifying whether we should mark the tuples for delete or we do a purge delete.

TestCase File Name : DeleteQuery

Interface Implementation :

Implementation of delete-query is similar to select query. A scan object is obtained and it will be traversed to get tid of the corresponding tuples which should be deleted. Next this tids are used to mark tuples for deletion and then can be deleted using purge delete function.

IV. ANALYSIS OF THE OUTCOMES

We've the following schema for our table:
coln1: A (CHAR[25])
coln2: B (CHAR[25])
coln3: C(INT)
coln4: D(INT)

We've populated this table with 1024 records,

and buffer size is set to 100 (select queries) for the purpose of analysis. If we've changed any of these values for a specific test, we mention in the below test implementation.
We created the following test suite for the purpose of analysis. This test suite contains 4 type of tests, namely Batch Insert, Index Create, Select Queries, and Delete queries.

We've selected the following set of queries for analysis:
Query 1: Batch Insert
Query2: Create Btree index on a column
Query 3: Create Bitmap index on a column
Query 4: Select A,B,C,D from table
Query5: Select D from table where C=6 (btree and bitmap indexes created on C)
Query 6: Select C from table where C=6 (btree and bitmap indexes created on C)
Query 8: Select A,B from table where C>6 (btree index created on C)
Query 9: Delete from table where C=6 (with purge)
Query 10: Delete from table where C=6 (without purge)

Batch Insert

For batch insert, we use the query batchinsert DATAFILENAME COLUMNDBNAME COLUMNARFILENAME NUMCOLUMNS, and we received the below results, for different number of input records:

No. of records	1k	6k	12.5k	25k	30k	50k
No.of reads	92	488	966	2004	2444	859021
No. of writes	92	488	966	2004	2005	118156

Table 1: Batch Insert Result

There's a huge increase of the number of reads and writes from 30k records to 50k records. This occurs because, at this point there are a lot of page faults, and hence a lot of pages reads will happen. The default buffer number was increased to 100000. This was done because we were getting buffer out of space exception for 50k records, when the buffer size was set to 1000.

Index Create

For Index create queries, we create btree index and bitmap index. The create index query is as follows:
index COLUMNDBNAME COLUMNARFILENAME COLUMNNAME INDEXTYPE.

Here in IndexType we specify whether the index is either btree or bitmap. Btree index is created on top of the columname provided, and bitmap index is created on all the unique values on the column provided.

We created both btree and bitmap indexes on the column C and verified the number of reads and writes performed. This is mentioned in table 2.

Index Type	Reads	Writes
Bitmap	37	24
Btree	48	35

Table 2

While creating a btree and bitmap, the columnar file is opened and a full scan is performed to read the records. Bitmap is a linked list of BM pages as mentioned earlier. A bitmap index is created for all the unique values, and reads are performed across all the pages in this columnar file.

For Btree, we use the minibase implementation.

Select-Query

In Select Query, we perform the below queries:

(1). Select A,B,C,D from table

(2). Select D from table where C=6 (btree and bitmap indexes created on C)

(3). Select C from table where C=6 (btree and bitmap indexes created on C)

(4). Select A,B from table where C>6 (btree index created on C)

1024 records were inserted using batch inserts. Btree index and bitmap index is created on top of the column C, using index create queries. The different accesstypes used are Filescan, Columnscan, Bitmap, and Btree.

The number of page reads and writes for the above queries are plotted in Chart 1.

Query 1: SELECT testColumnDB students A,B,C,D 100 <ACCESSYPE>

In this, we request for all the columns A, B, C, D from the students table. Here, there are no condition which needs to be matched. So all the access types will have the same number of page read counts, irrespective of the indexes. We got a result of 91 page reads. and 0 writes.

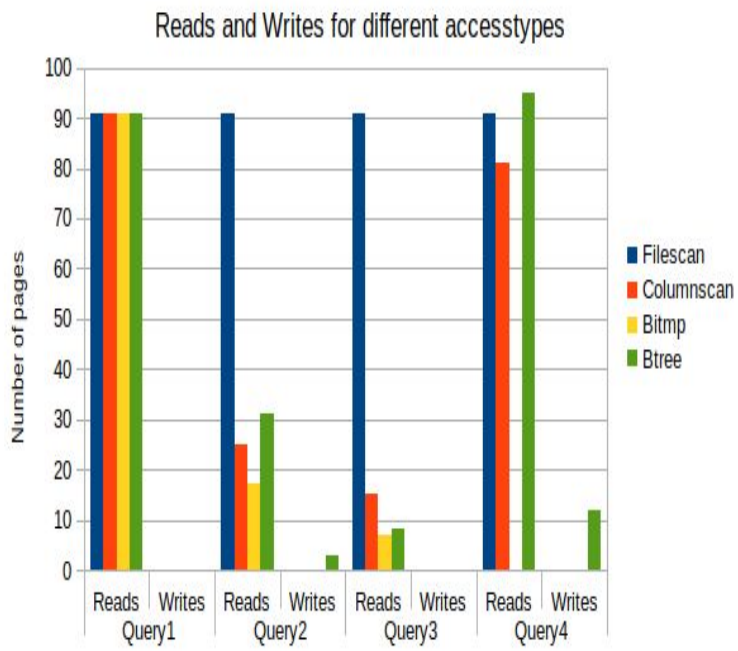


Chart 1: Number of page reads and writes for different accesstypes.

Query 2: SELECT testColumnDB students D {C = 6} 100 <ACCESSTYPE>

In this query, we request for the data from column D, where C=6, and below are our observations for various Access types:

File Scan: File scan is a simple scan of all the columnar files and return all the values in column D which matches the criteria. Here, the number of reads were 91 similar to query 1 without any condition.

Columnfile scan: In this accesstype, we scan the columnar file on column D and column C and returns all the values in D which matches the condition C=6. The number of page reads were 25, which is reduced from the normal file scan. This is because, we scan through only the columnar file D and C, rather than scanning through all the files.

Bitmap scan: In this accesstype, a bitmap index is created on column C. When the select query is issued, using the bitmap index for C=6, the values, for all the positions with

value 1 selected by bitmaps, are returned for column D. The number of page reads were reduced to 17. This is because, we didn't scan through the columnar file of C, instead we open only the bitmap index, and additionally, all the records in column D were also not read.

Btree scan: In this accesstype, a btree index is created in column C. When the select query is issued, using a scan on the btree index C, the corresponding RIDs are returned and using the corresponding RIDs of column D, the values are returned. The number of page reads are 31, and writes are 3. All the Leaf pages and the corresponding pages from column D are returned. We used the exisiting minbase implementation.

Query 3: SELECT testColumnDB students C {C = 6} 100 <ACCESSTYPE>

In this query, we select all the values of the column C, where C=6. Here, a btree index and bitmap index is created on the column C. The different access types are:

File scan: This is similar to the earlier file scans. A simple scan of all the columnar files and return all the values in column C which matches the criteria. Here, the number of reads were 91 similar to query 1 without any condition.

Columnfile Scan: In this scan, all the column values of C, which matches the criteria C=6 are returned. Here the number of page reads are 15. This is less than that of query2. This is because, in this scan, we are reading all the pages of column C for filtering and these files will be present in the buffer. So when returning the records, the number of page reads are really less.

Bitmap Scan: Similar to query 2, a bitmap is created on column C and the values for C=6 are returned for all the positions which matches the criteria. The number of page reads are 7, which is less than that of query 2. This is because, only the BM pages are accessed in this query. The values are returned by reading the BMpages alone

Btree Scan: Similar to Btree scan of query 2, a btree index is created on column C. The number of page reads are only 8, which is less than that of query 2. This is because this is an index only query, and all the values are returned from the index only. The data pages are not accessed here.

Query 4: SELECT testColumnDB students A,B {C > 6} 100 <ACCESSTYPE>

In this query, we select all the values from column A and B where C=6. Here, we have btree on column C and bitmap index on C=6. The access types are:

File Scan: Similar to the above queries, here we scan through all the columnar files and the values from columns A, and B are returned where C=6. Here, the number of reads were 91 similar to query 1 without any condition.

Columnfile Scan: In this scan, all the values of columns A and B are returned, where C=6. The number of page reads are 81. This is higher than query 2 and 3. This is because we're scanning through 3 complete columns and return the values from those positions where C=6.

Bitmap Scan: We're performing a bitmap operation on a range query as per assumptionsi [9].

Btree Scan: This access type is similar to query 2. Here, a btree scan is performed through the bree index of C, and the RIDs which matches C>6 are returned, and the corresponding records from column A and B are returned. The number of reads were 95. This is because, the btrees have 67% utilization and hence the number of pages to be accessed are high.

Deletefile query:

The delete works by calling the select query and then with the Tuple ID received we delete it. The delete works by finding the current pointer and by shifting the bits one position to left. The reads exceeds write in filescan, columnscan but is very evident in Btree scan and bitmap scan because the index and the data file should be changed. The data for reads and writes against the access pattern is displayed in table 3, and chart 2.

As mentioned in assumption (14), the delete using Btree implementation is not working correctly, so the number of read and write pages are really less.

For the bitmap query access type, we perform a reorganization of bitmap, whenever a purge is performed. This is to ensure that the bitmap index points correctly to the record.If all the entries in a data page is deleted, this page will be deleted from the page directory. If we don't rearrange the bitmap, then we won't be able to capture these changes.

Reads	Writes	Query Access patten
176	93	Filescan
222	97	BitMap
108	8	Btree
178	94	Column scan

Table1: Reads and Writes for deletion

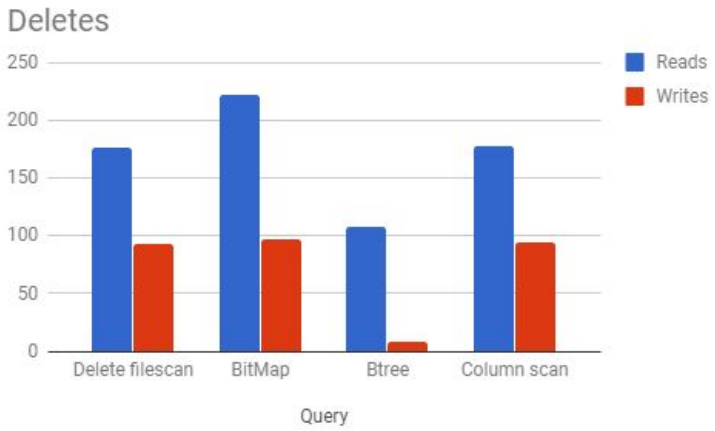


Chart 2: **Reads and Writes for deletion**

V. SYSTEM REQUIREMENT SPECIFICATION

The minimum and recommended system configuration required for running this project are as follows:
Operating System: Windows 10 onwards
Processor: Intel Core i5 @ 1.7 GHz
RAM: 8GB
Graphics RAM: 2GB System Type: 64-bit operating system, x64 bit based processor
Software Requirements: Java version "1.8.1"

VI. RELATED WORK

1. **Paper 1:** Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. **Column oriented Database Systems**. Tutorial at VLDB'09 **Overview:** This paper deals with simulation of column stores on top of row stores. This paper enforces the fact that c-stores on top of row stores will not be able to harness all the advantages of column stores. This is due to the difference in the way how data is stored and executed. This paper deals with techniques like late materialization which improves the performance by a factor 3. It also

discusses how compression and new-join algorithm can also be used to improve performance.

2. **Authors:** Anuradha S. Kanade, and Dr. Arpita Gopal,
   Name: "Choosing Database System, Row or Column Store",
   Published in: Information Communication and Embedded Systems (ICICES), 2013 International Conference on
   Overview: This paper presented a study which identifies the problems in traditional relational database system, and points out the benefits of using a columnar database system in the fields of business intelligence and analytics domain. The simplicity of the column store approach is really good in the fields mentioned. Some advantages of column store over row store include, Query optimization, compression, analytic performance, parallelization etc.
3. **Authors:** Arpita Gopal, and Vandana Bhagat
   Name: Comparative Study of Row and Column Oriented Database
   Published in: Emerging Trends in Engineering and Technology (ICETET), 2012 Fifth International Conference on
   Overview: RDBMS data warehouse are difficult to maintain and design for large amount of data, inefficient for their use of disk space and I/O. Columnar databases use less disk space and are more efficient in their I/O demands than records-based data warehouses but they have their own compromise between optimizing for new record insertion versus data selection and retrieval.Due to its column based architecture and its compression algorithms it stores data in archive format and reduce the volume of physical storage requirement by 20–40 percent.

VII. CONCLUSION

In this phase of the project, we've implemented column stores on top of row stores. Columnar databases store data column-wise. We've implemented bitmap and btree indexes on the columnar files and could observe that the records were fetched efficiently through these querying schemes. The analysis test suite consists of various queries that cover the different querying types. With this analysis, we can conclude that the presence of bitmap and btree indexes have a great impact on the overall performance of the database. This gave us an opportunity to understand the database implementation concepts.

VIII. BIBLIOGRAPHY

[1]. https://en.wikipedia.org/wiki/Database
[2].https://en.wikipedia.org/wiki/Data_model
[3].https://en.wikipedia.org/wiki/Database_schemal
[4].https://en.wikipedia.org/wiki/Relational_algebra

[5].https://en.wikipedia.org/wiki/Relational_calculus

[6].https://en.wikipedia.org/wiki/SQL

[7].http://research.cs.wisc.edu/coral/minibase/bufMgr/bufMgr.html

[8]..https://www.techopedia.com/definition/3854/parser

[9].https://www.techopedia.com/definition/26224/query-optimizer

[10].http://pages.cs.wisc.edu/~dbbook/openAccess/Minibase/spaceMgr/heap_file.html

[11].http://pages.cs.wisc.edu/~dbbook/openAccess/Minibase/spaceMgr/dsm.html

[12].https://en.wikipedia.org/wiki/B%2B_tree

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

[14].https://www.geeksforgeeks.org/indexing-in-databases-set-1

[15].https://en.wikipedia.org/wiki/Bit_array

[16].https://en.wikipedia.org/wiki/Bitmap_index

APPENDIX

CONTRIBUTIONS

A. ***MEMBER 1 -JITHIN PERUMPARAL JOHN***
ColumnarFile, ValueClass, Report, Analysis

B. ***MEMBER 2 - ARCHANA RAMANATHAN SESHAKRISHNAN***
Interfaces, TID, ColumnarFile, Report

C. ***MEMBER 3 - SOUMYA ADHYA***
Interfaces, ColumnDB, BM, Report

D. ***MEMBER 4 - DIXITH KUMAR KURRA***
Bitmap, ColumnIndexScan, PCounter, Report

E. ***MEMBER 5 - ADITYA CHAYAPATHY***
BitMapHeaderPage, BMPage, BitMapFile, Report, Analysis

F. ***MEMBER 6 - EJAZ SAIFUDEEN***
ColumnarFile, TupleScan, ColumnarFileScan, Report