CSE – 510 PROJECT PHASE III

IMPLEMENTING QUERY OPERATORS IN COLUMN-ORIENTED DATABASE MANAGEMENT SYSTEM

(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 deals with implementing query operators on column stores imitating row stores. In this phase, we have implemented the following operators: ColumnarIndexScan, ColumnarNestedLoopJoin, ColumnarBitMapEquiJoin and ColumnarSort. Since we are simulating column stores, we can make use of some of the operators like ColumnarBitMapJoin and ColumnarSort in order to efficiently answer queries. We perform columnarSort with buffers as low as 3. 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.

Keywords: row-store, column-store, equality search, range search, relational DBMS, bitmap index, b tree index, query operators, nested loop join operator, sort operator, bitmap equijoin operator, index scan operator

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.

In this phase, we focus on implementing the columnar Index scan, Nested Loop joins, Bitmap equijoins, and Sorting operators. We generalize the columnar Index scan implemented on phase 2, by providing one or more index file and returns matching tuples. This is also generalized for inequality searches. Furthermore to the previous phase, we're implementing joins on two columnar files. The joins we implement are nested loop joins to join two columnar files and bitmap equi joins, for joining two columnar files using bitmap index. In addition to that, we're performing sorting on a given columnar file for a given condition. Sorting needs to be implemented using buffers as low as 3.

- **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. While performing Columnar sorting, the column which we use for sorting condition should be in the projection list.
5. While performing Bitmap Equijoin, we assume that enough number of buffer pages are available to bring the necessary bitmap files into the buffer.
6. We should have at least 3 buffer pages available for sorting. This doesn't take into account the buffer used by the relation we are trying to sort.
7. Btrees do not differentiate between > and >=.
8. We use a sort merge operation for index scans, so, we assume we have enough buffers to perform the sort.
9. In the case of Index scan, only AND operation is supported when scanning on multiple indexes. Eg: (a=5 v a=6)^b=7 is valid. (a=5)v(b=7) is not.

- **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]

EquiJoins: Equi joins performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as

comparison operator in the where clause to refer equality. [17]
Runs: Each sorted subfile is referred as a run.
Joins: A join clause is used to combine rows from two or more tables, based on a related column between them [21].

II. IMPLEMENTATION

COLUMNAR_INDEX_SCAN

Purpose:
Scans the given columnar file, given one or more index files and return the matching tuples.

Implementation:
The challenging point in the implementation of this function is the selection of matching tuples from different scan objects. This is done by using get_next_position() function which propagates from 0 to n-1 heapfiles (depending on the number of columns in the table) only if the preceding position results match. The working is very similar to sort-merge join which interleaves scan results by taking the advantage of the sorted order of the incoming data stream.

Methods:

- **ColumnarIndexScan:**

– takes the columnarfile name, column numbers to scan, index type on them, selection predicates, selection condition for columns with no indexes, a boolean variable to identify if it is index only or not, columns on which the select conditions to be applied, Field specifications for output files.

- **get_next()**

- This function in turn calls get_next_position(), where the actual logic resides. This get_next_position sets the current position seen as the max _so_far variable's value if the current position's value is greater. When the current max position is present in all the scan objects it gets returned. The sorted ordering of the get_next function of the iterator is used to efficiently prune some of the outcomes. In order to implement this function, all the iterator must return the positional value. Hence Btree is modified to return the positions in sorted order.

- **close()**

- This function closes all the iterator objects.

COLUMNAR_NESTED_LOOP_JOINS

Purpose:
To perform Nested Loop joins on columnar files.

Implementation:
The implementation of this function is very similar to the traditional nested loop join except for the parts where heapfile scan is called columnar file scan is called. There is also the use of tid in place of tid. The general logic is the application of conditional expression on the outer relation and the application of the predicate on the inner relation, and then finally joining them both by applying output filter on both the set of satisfying tuples.

Methods:

- **ColumnarNestedLoopsJoin()**

-The constructor is used to initialize various member variables such as array of input attributes, their lengths, number of output fields, their fldspecs, amount of memory, right filter and output filter, left filter in the form of iterator, columnar file name.

- **get_next()**

This function is used to return a stream of tuples satisfying the join condition. We pass

in the iterator for the outer relation, right filter for the inner relation and the output filter for the tuples that are emerging out of the iterator and the right filter.

- **close()**

- This function closes all the iterator objects.

COLUMNAR_BITMAP_EQUIJOIN

Purpose:
To join two columnar files using bitmap indexes on the given field.

Implementation:
Input:

1. Equijoin condition
2. Outer relation condition
3. Inner relation condition

Output:

1. Joined tuples that satisfy the above 3 conditions

Steps Involved:

1. For each equijoin predicate, obtain the set of values that are common for both the attributes. The bitmaps of only these values are relevant for the equijoin operation with respect to the current predicate. One set of common values is maintained per predicate.

2. For each predicate, we obtain the list of positions from both the relations that satisfy the given predicate. This is done by obtaining the bitmap (for a given common value) and joining the positions that have the bit set to value 1. This gives us a set of positions per predicate.

3. The last step involves combining the set of predicates (obtained in the last step) to get a final list of predicates that satisfy the entire equijoin predicate. The predicates that involve the "AND" operation are combined using intersection operation. The predicates that involve the "OR" operation are combined using union operation.

The final list of positions are subject to the inner and outer conditions. Once passed, the necessary attributes are projected.

Methods:

- **extractUniqueValues(int offset, HashMap<String, BitMapFile> allBitMaps)**

This method extracts the unique values in given column from the available bitmap files

- **nestedLoop(List<List<Integer>> uniqueSets)**

Given the unique sets between two columns, this method performs the nested loop on the bitsets. The cols on which this operations needs to performed is extracted from the join expression given from the user

- **getSerializedConditionList(CondExpr[] condExprs)**

Given a conditional expression array in cnf form, returns a list of operations in the cnf form. This list is used in processing of the query to infer the kind of operation that needs to perform between the conjucts

COLUMNAR_SORT

Purpose:
To perform sorting on the given columnar file based on the provided conditions.
We've created a class **runfile** to handle the sorted runs during each pass. We're using the sorted runs implementation of minibase.

Implementation:
In this implementation of sorting, we are using temporary heap files without the directory structure. Sorting operation takes a condition on a columnar file whether to sort the file ascending or descending. In addition

to that, we could also limit the results projected using a predicate condition on any fields.

As studied in class, we could handle sorting with buffer pages as low as 3 by performing multiple passes on the runs generated in pass 0.

Methods:

- **setup_for_merge**(int tuple_size, int n_R_runs):

This function is used for setting up the merging of runs. This function open an input buffer for each run and store the first element from each run to the heap. The function delete_min(), will get the minimum of all runs.

- **generate_runs**(int max_elems, AttrType sortFldType, int sortFldLen):

This function generate the sorted runs using heapsort. It keeps track of the maximum number of runs, and it maintains a fixed number of runs in the heap.
Once the queue is full, this function start writing to file, and it also tries to add new tuples to this queue. All the tuples that does not fit to this queue is added to another queue temporarily. Once we could not add more tuples to this run, this function writes to file, and these steps are repeated for the tuples in the temporary queue.
This is repeated until we have no tuples to write out.

- **Tuple delete_min():**

This function is used to remove the minimum value among all the runs. Once we remove a tuple of a run from the queue, we'll replace this with another tuple of the same run in the queue.

- **MIN_VAL**(Tuple lastElem, AttrType sortFldType):

This function set the lastElem to be the minimum value of the corresponding type.

- **MAX_VAL**(Tuple lastElem, AttrType sortFldType): This function set the lastElem to be the maximum value of the corresponding type.
- **get_next()**

This function returns the next tuple in the sorted order

- **close()**

This function performs the cleaning up operation. It release the buffer pages from the buffer pools and remove temporary files from the database.

III. INTERFACE SPECIFICATION

Following the specification document different interfaces were implemented to test Sorting, BitMap EquiJoin and Index loop Join. To test ColumnarIndexScan a function to initiate a ColumnarIndexScan was added to Select Query interface as it is a generalized scanning method capable of handling indices on multiple columns with different comparison conditions like equal, not equal, smaller than and greater than.

ColumnarIndexScan :

Parameters : 1. Name of columnar file where the relation is stored 2. Array of column numbers on which index should be created 3. Array of index types indicating the Type of the indexes on the columns specified in the last step 4. Array containing sizes of the string fields 5. Number of columns which will be used to scan the relation and apply selection condition 6. Number of fields which will be

projected in the output. 7. Specification of the fields which will be provided as output 8. Array indicating conditional expression based on which this index scan will give output. 9. Whether the scan is index only or not. 10. Sort Memory

Test Case File Name : SelectQuery.Java

Interface Implementation :

Query: **Select** **COLUMNDB COLUMNFILE PROJECTION OTHERCONST SCANCOLS [SCANTYPE] [SCANCONST] TARGETCOLUMNS NUMBUF**

eg: *Select testColumnDB columnarTable A,B,C "C = 5" A,B [BTREE,BITMAP] "(A = 5 v A = 6),(B > 7)" A,B,C 100 100*

In this test case we generalise the selection query so that it can handle multiple indices and different comparison operators. We added a generalized method called "processRawConditionExpression" in InterfaceUtils.Java which will scan the condition string to retrieve the column numbers on which the indexing will be done and the conditions based on which selection query will work respectively. We get the selection conditions and extra constraints as a conditional Expression array. We then use this conditional expression array along with the projection columns and column types to initialize a Columnar IndexScan object. To display the output of this scan we use get_next method of scan object and print out the results.

BitMapEquiJoin :

Parameters : 1. Name of columnar db where the inner and outer relation is stored 2. Name of the outer relation file 3. Name of the inner relation file 4. Constraint on the outer relation file 5. Constraint on the inner relation file 6. Equality constraint 7. Target Columns 8. Number of Buffers 9. Sort Memory

Test Case File Name : BitmapEquijoins.Java

Interface Implementation :

Query: **COLUMNDB OUTERFILE INNERFILE OUTERCONST INNERCONST EQUICONST [TARGETCOLUMNS] NUMBUF**

eg: *testColumnDB columnarTable1 columnarTable2 "([columnarTable1.A = 10] v [columnnarTable1.B>2])^([columnnarTable1.C = 20])" "([columnarTable2.A = 10] v [columnarTable2.B > 2]) ^ ([columnarTable2.C = 20])" "([columnarTable1.A = columnarTable2.A] v [columnarTable1.B = columnarTable2.B]) ^ ([columnarTable1.C = columnarTable2.C])" columnarTable1.A,columnarTable1.B,columnarTable2.C,columnarTable2.D 100 100*

In this test case we retrieved the inner join constraints as a conditional expression using processRawConditionExpression method in InterfaceUtils. We also used the same methods to retrieve the outer join constraints as conditional expression.

We also added a method called processEquijoinConditionExpression to retrieve the equijoin condition as a condition expression.

After we get conditional expression for inner, outer relation and equi join condition we pass these along with outer and inner columnar file to initialize a ColumnarBitmapEquiJoin object. Which automatically prints out the tuples satisfying the joining constraint.

ColumnarNestedLoopJoin:

Parameters:

ColumnarDataBaseName,columnarFile1, columnarFile2,projection, outerConstraints,outerScanColumns, outerScanTypes,outerScanConstraints, outerTargetColumns,InnerConstraints, innerTargetColumns,joinConstraints,bufferSize.

Test Case File Name : ColumnarNestedLoopJoin

Query: ***COLUMNDB CF1 CF2 PROJECTION OUTERCONST OUTERSCANCOLS [OUTERSCANTYPE] [OUTERSCANCONST] OUTERTARGETCOLUMNS INNERCONST INNERTARGETCOLUMNS JOINCONDITION NUMBUF***

Eg: *testColumnDB columnarTable1 columnarTable2 "columnarTable1.Attr1,columnarTable2.Attr2" " " " " "FILE" " " "A,C" " " "A,C" "columnarTable1.Attr3=columnarTable2.Attr3" 20 100*

This interface is dedicated for Columnar nested loop joins which would join two columns based on the nested conditions. It works by checking each satisfying tuple of the outer relation with each of the inner relation's tuples. The interface calls the NestedLoopJoin class with join condition, Right (inner) condition and Output condition. The condition for the outer relation is passed in the form of a suitable Iterator.

ColumnarSort:

Parameters : 1. Name of columnar db where the file which needs to be sorted is stored 2. Name of the relation file to be sorted 3. Name of the column based on which the replation must be sorted 4. Order of the sorting (Ascending or Descending) 5. Number of Buffers 6. Number of Minimum pages to be used 7. Sort Memory

Query: **Sort COLUMNDB COLUMNARFILE COLUMNNAME SORTORDER BUFFSIZE**

Eg : SortInterface testColumnDB columnarTable columnName ASC 100 3 100

Test Case File Name : SortInterface.Java

Interface Implementation :

Query: **Sort COLUMNDB COLUMNARFILE COLUMNNAME SORTORDER BUFFSIZE**

Eg : SortInterface testColumnDB columnarTable columnName ASC 3 100

To test sorting we initialized a columnarfile scan on the file provided as input parameter. We pass all the columns as projection output to the columnar file scan as sorting query should display all the columns as output.

We then use this columnarfilescan object and use it along with sorting column number, type of the sorting column number and Sort order to initialize a columnarsort object.

We display the sorted output by scanning the outputs using get-Next() method on columnarsort object.

IV. OUTCOMES

ColumnarIndexScan

For columnar Index scan, we provide the list of projected columns, conditions on the column which doesn't have any index, columns with their corresponding indexes and the selection condition on the indexes.

The output of the query will be the set of tuples with the projected columns and which satisfies the given conditions on non-index column and index columns.

ColumnarNestedLoopJoins

This function joins the relations to get an intermediate result. This table contains the joined output of both the tables. Firstly the two tables are checked to see if they can be joined. If that is true, then the results are joined on a tuple by tuple basis.

ColumnarBitmapEquiJoins

In Bitmapequijoins, we've 3 conditions:
Equijoin condition, inner relation condition and outer relation condition.
The output is a joined tuples which matches these 3 conditions.

ColumnarSort

In ColumnarSort, we sort the tuples based on a condition on a given column, and the result is sorted tuples.

In addition to this, we could also specify a condition on any of the columns which would return only the tuples in sorted order, which matches the condition.

V. SYSTEM REQUIREMENT SPECIFICATION

The minimum and recommended system configuration required for running this project are as follows:
Operating System: Ubuntu 16.04 onwards / 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

Paper 1: David J.DeWitt, Jeffrey F. Naughton, Joseph Burge **Nested Loop Joins Revisited** Published at Parallel and Distributed Information Systems, 1993.
Overview: Research Community has considered hash based parallel algorithms as a choice for a period of time. But, none of the commercial parallel dbms uses this join, instead they use nested loops with index or sort-merge. This paper presents a comparison study of parallel nested loop join with index and parallel hash based index. The outcome of the study was though parallel hash based index joins outperforms nested loop for most combination of input, there are certain cases when nested loops with index provide better performance, for example when the relation sizes are very different. This paper concludes that parallel database systems could profit from implementing both of them.

Paper 2: 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.

VII. CONCLUSION

In this phase of the project, we've implemented columnar Index scan on multiple index files which return matching tuples. In addition to that, we've also implemented Nested Loop Joins on Columnar files and Equijoins using Bitmap indexes. For Nested Loop join, we could implement different join condition including equality, inequality etc. Finally, we implemented multi pass sorting. We could use buffers as low as 3 buffer pages to complete the sorting. A set of sorted runs will be created in the first pass, and then merging is implemented. This gave us an opportunity to understand the database implementation concepts.

VIII. BIBLIOGRAPHY

[1] Databases, retrieved from https://en.wikipedia.org/wiki/Database

[2] Data Mode, retrieved from https://en.wikipedia.org/wiki/Data_model

[3] DataBase Schema, Retrieved from https://en.wikipedia.org/wiki/Database_schemal

[4]Relational Algebra, retrieved from https://en.wikipedia.org/wiki/Relational_algebra

[5] Relational Calculus , retrieved from https://en.wikipedia.org/wiki/Relational_calculus

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

[7] Buffer Manager, retrieved from http://research.cs.wisc.edu/coral/minibase/bufMgr/bufMgr.html

[8] Parser, retrieved from https://www.techopedia.com/definition/3854/parser

[9] Query Optimizer, retrived from https://www.techopedia.com/definition/26224/query-optimizer

[10] Heap file, retrieved from http://pages.cs.wisc.edu/~dbbook/openAccess/Minibase/spaceMgr/heap_file.html

[11] Space Manager, retrieved from .http://pages.cs.wisc.edu/~dbbook/openAccess/Minibase/spaceMgr/dsm.html

[12] Btree, retrieved from https://en.wikipedia.org/wiki/B%2B_tree

[13]Column Oriented DBMS, retrieved from https://en.wikipedia.org/wiki/Column-oriented_DBMS

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

[15] BitArray, retried from https://en.wikipedia.org/wiki/Bit_array

[16] Bitmap index, retrieved from https://en.wikipedia.org/wiki/Bitmap_index

[17]Equi joins, retrieved from https://www.w3resource.com/sql/joins/perform-an-equi-join.php

[18] Nested Loop Joins, retrieved from http://pages.cs.wisc.edu/~dewitt/includes/paralleldb/nestedloops.pdF

[19] Column-Oriented Database Systems, retrieved from http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf

[20] Database Management Systems, R. Ramakrishnan and J. Gehrke. McGraw-Hill, Third edition

[21] Joins retrieved from https://www.w3schools.com/sql/sql_join.asp

APPENDIX - CONTRIBUTIONS

MEMBER 1 -JITHIN PERUMPARAL JOHN
ColumnarSort, OutputScript, Interface, Report

MEMBER 2 - ARCHANA RAMANATHAN SESHAKRISHNAN
ColumnarIndexScan, ColumnarNestedLoopJoin, Report

MEMBER 3 - SOUMYA ADHYA
ColumnarSort, OutputScript, Outputfile, Report

MEMBER 4 - DIXITH KUMAR KURRA
ColumnarBitmapEquiJoin, Interface, Report

MEMBER 5 - ADITYA CHAYAPATHY
ColumnarBitmapEquiJoin, Interface, Report

MEMBER 6 - EJAZ SAIFUDEEN
ColumnarIndexScan, Interfaces, ColumnarSort, Optimization, Report