Volume 4, Issue 4, April 2014 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

A Framework for Supporting Mixed-Join Queries in Cloud Data Stores

Amira Elzeiny [*], Ahmed Abo Elfetouh, Alaa Riad
Information System Department,
Faculty of Computer and Information Sciences,
Mansoura University, Egypt

Abstract— Cloud storage provides high availability and scalability for web applications, but it loses the power of complex queries. The cloud data stores may include row oriented tables and column oriented tables in the same database. The cloud storage systems must be capable of performing mixed queries in these tables with no need for converting the tables' orientation. This paper presents a framework for supporting mixed join queries in cloud data stores. Mixed join queries refers to joining data from two tables with different orientations; column and row, without performing any conversion between the tables' orientations. This means that the join operation is executed upon the tables in its native orientation. The proposed framework aims at enhancing the query processing of relational data in the cloud storage systems by removing the overhead associated with tables' orientations conversions, especially that the conversion overhead for mixed join in the cloud is usually not trivial from the perspective of performance and memory consumption.

Keywords— Cloud data stores, column-oriented tables, Join algorithms, Query support framework, Mixed join queries

I. INTRODUCTION

There are two ways to map a two-dimensional relational database table onto a one-dimensional storage interface; store the table row-by-row, or store the table column-by-column. Historically, database systems implementations and research have focused on the row-by row data layout, since it performs best on the most common applications for database systems specially business transactional data processing [1]. However, there are a set of emerging applications for database systems for which the row-by-row layout performs poorly.

These column oriented database systems have been shown to perform better than traditional row-oriented database systems on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications. Generally data must be accessed in a table in the same way it was stored. So performing queries on a database requires dedicated query operators that can access a specific type of a storage model.

So the question is: Consider the following case; what if the database in the cloud is collected from different sources with different orientations? How will the cloud storage systems deal with this mixed database? It's difficult to deal with such database as it's not a column oriented or a row oriented as a whole. Can we get the advantages of the two orientations together? And, the next question is; do we have to convert the whole database tables to the same storage model to deal with? The answer is" no". In reality, a database having data stored in a column storage model may be asked to handle a transactional query relating to that data, and on the other hand, a database having data stored in a row storage model may be asked to handle an analytical query relating to that data. For example; a cloud database where the data was stored in a row oriented format may receive a mixed set of queries requiring both transactional and analytical queries. In responding to such a mixed set of queries, a query engine may perform a mixed join operations.

This paper presents a framework for supporting mixed join queries on relational data in cloud data stores. Mixed join refers to joining data from two tables with different orientations; column and row, without performing any conversion between the tables' orientations. This paper is organized as follows; the next section introduces a background about the row and column orientations and their advantages, and gives an idea of the query processing in the cloud data stores, and discusses the different types of join algorithms. Section III discusses the related work. Section IV presents the new proposed framework. In section V the details of the mixed join queries algorithm is introduced. And finally section 6 concludes.

II. BACKGROUND

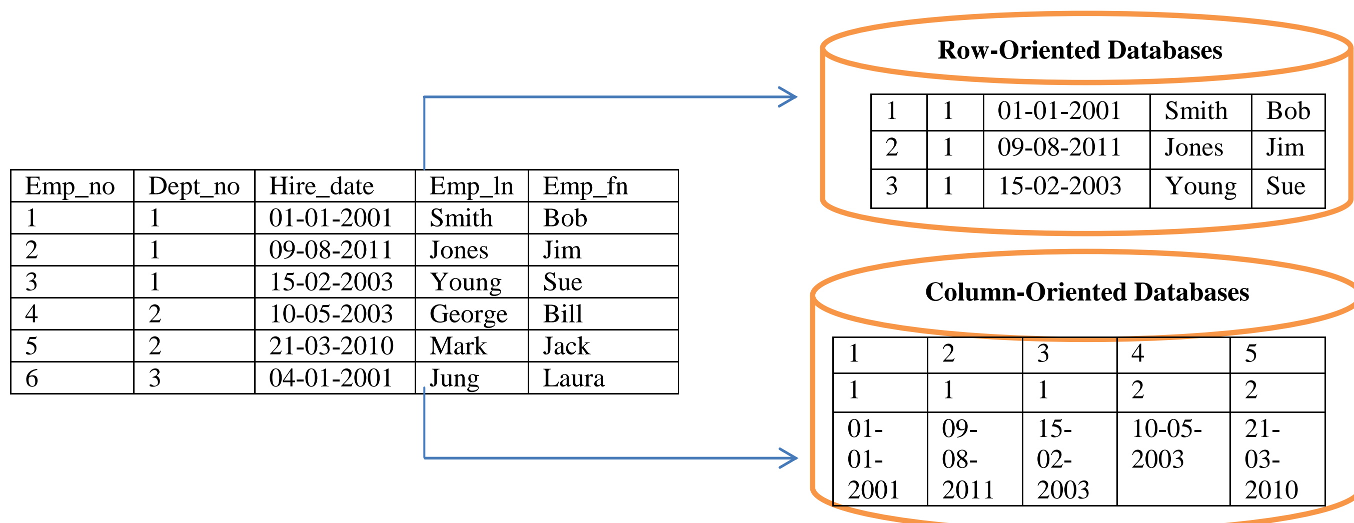
A. *Rows vs. Columns*

There are two ways to map database tables onto a one dimensional table: store the table row-by-row or store the table column-by-column. The row-by-row approach keeps all information about an entity together in one row. While the column-by-column approach keeps all attribute information together in one row.

Fig [1] shows an example of "Employees" table, and how it's represented in the two different orientations [2]. In this example, according to the row-by-row approach; all information about the first employee will be stored in the first row, and then all information about the second employee will be stored in the second row, etc.

© 2014, IJARCSSE All Rights Reserved

In the same example, according to the column-by-column approach; all of the employees numbers (Emp_no) will be stored in the first row, and the all the departments id's (Dept_id) will be stored in the second row, etc.



Emp_no	Dept_no	Hire_date	Emp_ln	Emp_fn
1	1	01-01-2001	Smith	Bob
2	1	09-08-2011	Jones	Jim
3	1	15-02-2003	Young	Sue
4	2	10-05-2003	George	Bill
5	2	21-03-2010	Mark	Jack
6	3	04-01-2001	Jung	Laura

Row-Oriented Databases

1	1	01-01-2001	Smith	Bob
2	1	09-08-2011	Jones	Jim
3	1	15-02-2003	Young	Sue

Column-Oriented Databases

1	2	3	4	5
1	1	1	2	2
01-01-2001	09-08-2011	15-02-2003	10-05-2003	21-03-2010

Fig [1] Row-oriented and Column-oriented Databases

The column-oriented database appears exactly like a traditional relational database; the logical concepts of tables, rows, and columns are the same, SQL commands are used to interact with the system, and most other RDBMS paradigms (e.g. security, backup/recovery, etc.) remain unchanged [21].

So, what are the actual benefits the column stores deliver over a legacy RDBMS with row-to-row data layout? Almost all the relational databases currently being offered are primarily designed to handle online transactional processing (OLTP) workloads [2]. A transaction maps to one or more rows in a relational database, and all traditional RDBMS designs are based on a per row paradigm. For transactional-based systems, the row-by-row architecture is well-suited to handle the input of incoming data.

However, for applications that are very read intensive and selective in the information being requested, the OLTP database design isn't a model that typically holds up well. Whereas transactions are row-based, most database queries are column-based. Inserting and deleting transactional data are well served by a row-based system, but selective queries that are only interested in a few columns of a table are handled much better by a column-oriented architecture.

On average, a row-based system does 5-10x the physical I/O that a column-based database does to retrieve the same information. As the physical I/O operations are the most expensive part of a query, and that an analytical query includes more rows of data than a query, the performance gap between row-oriented architectures and column-oriented architecture oftentimes widens as the database grows [21].

Column-stores are more I/O efficient for read-only queries as they read only those attributes which are accessed by a query. To get around their selective query inefficiencies, row-based RDBMS's utilize indexing, horizontal partitioning, materialized views, summary tables, and parallel processing, all of which can provide benefits for intensive queries, but each comes with their own set of drawbacks as well. For example, while indexing can certainly help queries complete faster in some cases, they also require more storage, impede insert/update/delete and bulk load operations (because the indexes must be maintained as well as the underlying table), and can actually degrade performance when they become heavily fragmented. Moreover, in business intelligence/analytic environments, the ad-hoc nature of such scenarios makes it nearly impossible to predict which columns will need indexing, so tables end up either being over-indexed (which causes load and maintenance issues) or not properly indexed and so many queries end up running much slower than desired [21]. But, a column-oriented database overcomes the query limitations that exist in traditional RDBMS systems by storing, managing, and querying data based on columns rather than rows. Because only the necessary columns in a query are accessed rather than entire rows, I/O activities as well as overall query response times can be reduced because it doesn't have to read an entire row to get the data needed. The end result for column databases is the ability to interrogate and return query results against either moderate amounts of information (tens or hundreds of GB's) or large amounts of data (1-n terabytes) in much less time that standard RDBMS systems can.

Column-oriented DBMSs may have additional advantages over row-oriented systems. Because column data are stored contiguously in column-oriented systems, and because such data may exhibit less entropy than data from different columns, higher data compression rates may be achieved in column-oriented systems than in row-oriented systems [23].

Column stores have many drawbacks [21] as well; the first is associated with the query performance when query involves several columns. Column stores can be faster than row-oriented stores when the number of columns involved in the query is small. This is dependent on a number of factors, including the number of columns, and the use of indexing. This is due to the I/O efficiency of the column stores for read-only queries since they only have to read from disk (or from memory) those attributes accessed by a query. But the performance degradation can be quite significant in the column stores as the number of columns increases, due to the re-composition overhead. If the query involves more than a few columns; either columns for retrieving data or columns for query predicates, the performance will be affected [22].

© 2014, IJARCSSE All Rights Reserved

The second drawback of the column stores is associated with inserting data; when we create a new data record, we are creating a data row. However, a Column Store does not have a row-orientation. Instead, a Column Store must decompose that data row into the individual column values, and store each of those column values individually. This adds up to a lot more block updates for a Column Store than a row-oriented store. This requires additional work for data insert and update operations.

B. Databases in Cloud Environment

The age we live in is a data age. Compared to the past, the amount of data or the size of the data used is much larger than before. The gap between the large amount of data being produced and the relatively limited size of traditional databases that are used to store the data challenges traditional database systems. Meanwhile, new requirements from data management are being called for: availability, reliability, and scalability [4]. Traditional approach of storing data locally in the user's hard drive is not able to cope with the changing requirements of users who daily deal with massive amounts of digital data, and hence need more scalability, high availability, and optimized resources allocation. It seems to have a limitation on handling such big data volumes and modern workloads. All of this and more led to the emergence of cloud data storage. It provides the users with all these capabilities and more.

Cloud Storage is an important part of cloud computing. Cloud Computing provides an opportunity to store data in Cloud Storage instead of storing it to computer's local hard drive. Users do not need to maintain large storage infrastructures. They can store data in remote data centers, controlled and managed by big companies like Apple, Microsoft, Google, and Amazon etc. Files saved in the cloud storage can be accessed from anywhere with any device with an Internet connection. Cloud storage can be considered as an online storage available on network hosted by third party vendors. Data is stored on virtualized pools of storage. It is delivered as a service on demand in a scalable and multi-tenant way [24].

The Cloud Database is constructed by collecting a number of sites. These sites are also called as nodes which are interlinked by a communication network. Every single node is a database class. Each database class has its own database, terminals, the central processor and their individual local database management system [8].

In cloud computing environment, data is distributed across a vast number of servers and costumers have little control on them, so an effective management of data is a big concern for organizations using cloud-based services. Data management applications are potential candidates for deployment in the cloud to get the advantages of using the cloud. This is because the cost of developing large database systems is high in both hardware and software. For many companies especially for start-ups and medium-sized businesses, the pay-as-you-go cloud computing model, along with having someone else worrying about maintaining the hardware and managing the database is very attractive [2].

Different Cloud storage providers use different storage architectures [25]. Fig [2] illustrates a generalized architecture of Cloud Storage as presented in [24].This architecture includes three layers:

1. Cloud Interface Layer
   It's a software layer provided by the cloud storage provider to connect cloud users to cloud storage service through Internet. This layer applies authentication and authorization techniques to authenticate the users.
2. Data Management Layer
   It's a software layer used to manage data of a particular cloud client. Data management is mainly concerned with activities like data storage, content distribution across storage location, data partitioning, synchronization, maintaining consistency, replication, controlling movement of data over network, backup, data recovery, handling millions of users, maintaining metadata and catalogue etc.
3. Storage Layer
   The Storage layer consists of two parts:
   - Virtualization: Storage virtualization gives illusion of unified storage. It maps distributed heterogeneous storage devices to a single continuous storage space and creates a shared dynamic platform. It is implemented by storage virtualization technology. Few virtualization technologies provide built-in availability, security and scalability to applications.
   - Basic storage: It comprises of database servers and storage devices of heterogeneous nature such as DAS, SAN, NAS etc.

The query processing in the cloud can be split into two phases [9]. The goal of the first step is to retrieve the cluster servers that may provide possible results to the query. Given a query Q, in order to guarantee the completeness of the result, all possible cluster servers that overlap with Q must be retrieved. To achieve this goal, many efficient algorithms are proposed to prune the search space.

While in the second phase, the query is routed to the servers and processed locally by the DBMS of the selected node.

Fig [3] shows the steps of the SQL query processing in cloud storage. The first step is parsing the SQL query; it translates the high level language query submitted by the user to the equivalent relational algebra expressions. The second step involves enumerating all possible plans for the given query based on the join conditions by the plan enumerator.

The next step is query optimization. As there are many equivalent transformations of the same high-level query and hence many possible execution plans, the aim of query optimization is to choose an efficient execution plan for processing a query. It chooses the one that minimizes the resources usage by using the information from the system catalog.

© 2014, IJARCSSE All Rights Reserved

The plan enumerator first lists out all possible ways of joining the input tables from the given SQL query, then calculates the estimated costs of the plans using the cost models and the estimation formulas. The chosen plan is then sent to the plan executer.
The best plan is then converted to a set of operations or jobs. These jobs are then executed on the clusters using the local DBMS. The specific cloud APIs are used for data access and stores on the cloud data storage.
The query plan contains such elements as joins, horizontal and vertical data merges, and select operations that are performed locally by the DBMS. Each element in the query plan has different algorithms of optimization.

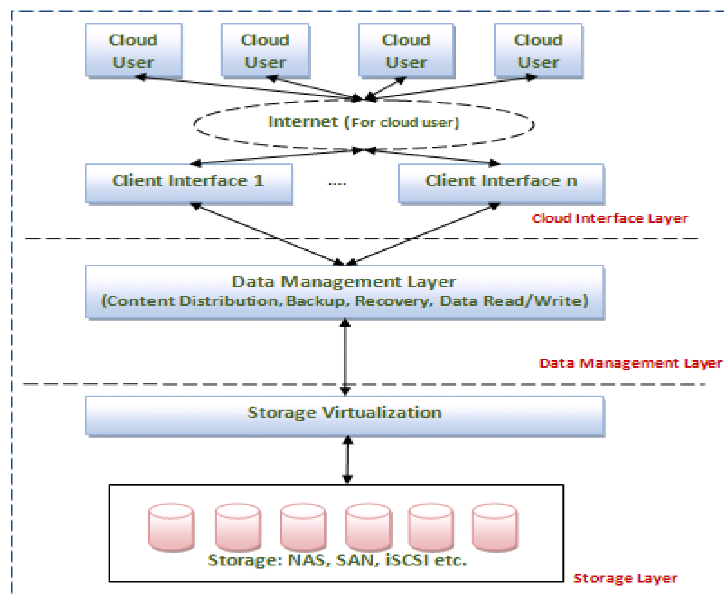


Fig [2] A Generalized architecture of Cloud Storage

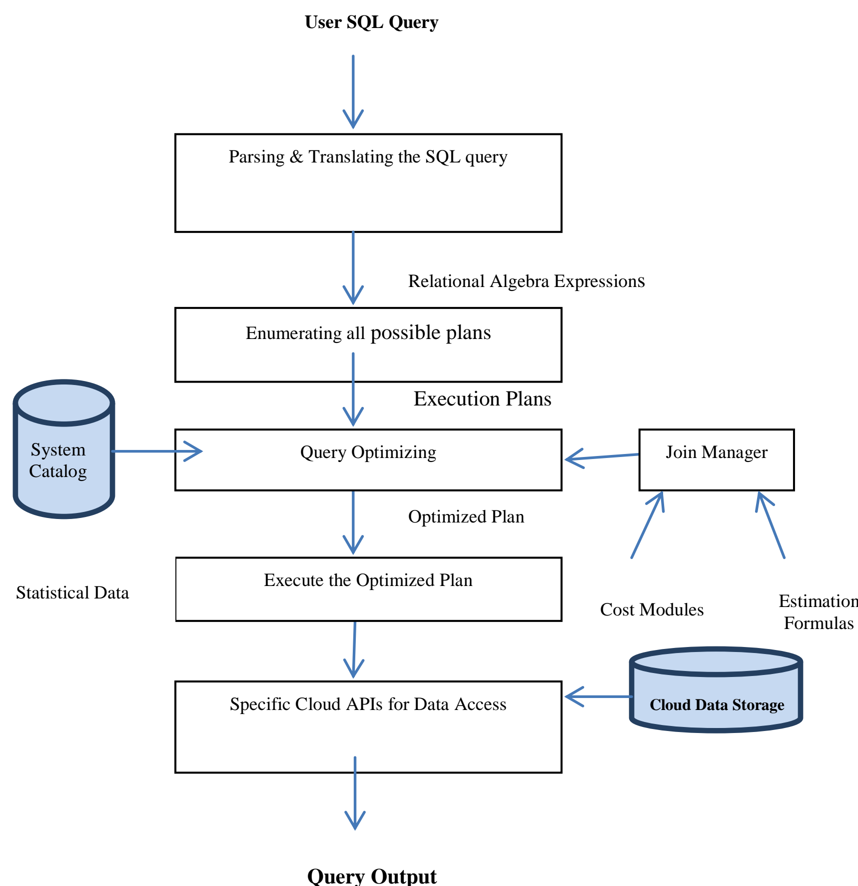


Fig [3] Steps of the SQL query processing in cloud storage.

© 2014, IJARCSSE All Rights Reserved

C. Join Queries

Join is a mean for combining fields from two tables by using values common to each. Join operation is considered as one of the fundamental operations on relational databases, and it is also one of the most expensive operations in any database system, and especially in the cloud storage systems. Its performance directly impacts the overall performance of the database systems, especially in an environment of massive data in which I/O cost dominates the execution time, just as in the case of cloud storage systems. Join queries, in the most general case, can range from simple operations; matching two records from different tables to very complex data-mining queries which scan the entire database for hours.

ANSI-standard SQL specifies five types of join: inner, outer, right, left, and as a special case, a table can join to itself in a self-join. Inner join is further classified into equi-join, natural join and cross join, and outer join is further classified as left outer join, right outer join and full outer join [28].

This paper considers only inner-joins which return all records that have at least one matching record, so the final combined record contains merged records from the concerned joined tables. Other types of join, such as outer-join (which may return records with no matching record), and semi-join (which only returns records from one table), are out of the scope of this paper.

Join algorithms may be categorized into three categories: 1) basic join algorithms, 2) join index based algorithms, and 3) architecture aware algorithms [7].

1) Basic join algorithms

Join processing has been researched extensively. Some classical join algorithms have been proposed, such as nested-loop join, sort-merge join, and hash join.

The Nested-loop join algorithm is considered as one of the simplest algorithms of join where for each record of the first table the entire records of the second table has to be scanned. This process is repeated for each and every record of the first table that is for all the first table records. The loop is of two levels and they are outer loop and the inner loop. First table loop is called as outer loop and the second table loop are called as inner loop. This algorithm has a repeated input/output scans of one of the tables, so it's considered as inefficient join algorithm.

The second algorithm is Sort-Merge join algorithm. This algorithm has two operations; sorting and merging. In the sorting operation; the two tables to be joined are sorted in ascending order on the join attribute. While in the merging operation; the two sorted tables are scanned sequentially, and the tuples of the two tables satisfying the join condition are merged and written to the result file. The Sort-Merge join algorithm is considered as efficient join algorithm when compared to Nested loop join algorithm.

The final basic join algorithm is Hash Based join algorithm. This algorithm is divided into partitioning phase and matching phase. During the first phase; the two tables are split into equal number of partitions, by the same hash function. The second table is matching, if any match is found, the two records are concatenated and placed in the query result. A decision must be made about which table is to be hashed and which table is to be matched. Since a hash table has to be created, it would be better to choose the smaller table for hashing and the larger table for matching.

2) join index–based algorithms

Index is a data structure that is used to speed up data retrieval. It has been used in practical applications extensively [26],[27]. A join index is an abstraction of the join of two tables. It specifies the surrogate pair of each join result. Surrogate can uniquely identify a tuple in table. Thus, a join index is a relation of two columns. By the available join index, there are some algorithms to process join operation.

Li et al. [16] present two algorithms Jive join and Slam join to process join using a join index. Two algorithms are duals of each another. Jive's join range-partitions the tuple ids of the second input relation, then processes each partition separately. Slam join forms sorted runs of tuple ids for the second input relation, then merges those runs.

Tan et al. [18] generalize traditional join index to a cluster-based join index. The objects in database are clustered according to their proximity. Instead of maintaining surrogate pairs of join results, a cluster-based join index represents cluster pair in which some members satisfy join condition. R-tree is used to determine the clusters which are related in cluster-based join index.

3) Join architecture-aware algorithms

Basic join algorithms do not take characteristics of the computer architecture into consideration, so they incur poor CPU utilization. Some researchers have proposed architecture-aware algorithms to improve data processing efficiency.

Shatdal et al. [19] propose cache-partitioning to split tables in partitioning phase as small as possible to make each partition of inner table fit in cache. However, it incurs high I/O cost because the partition number may be much high on massive data.

Chen et al. [20] exploit pre-fetching mechanism to improve hash join. They propose two new pre-fetching techniques: group pre-fetching and software-pipelined pre-fetching. The former modifies forms of compiler transformation to restructure the code such that hash probe accesses resulting from groups of consecutive probe tuples can be pipelined. The latter is used to avoid the intermittent stalls during the transition between groups.

By reviewing the former algorithms, some of them support joining in a database with tables stored in row storage model, while others support joining in a database with tables stored in column storage model, but none of these algorithms support mixed join in cloud environment.

© 2014, IJARCSSE All Rights Reserved

III. RELATED WORK

Conventionally, data must be accessed from a table in the same manner that it was stored. That is conventional computer storage techniques require dedicated query operators that can access specific types of storage models. For example, row query operators are used to process data stored in a database in row-formatted storage models and column query operators are used to process data stored in column-formatted storage models. Choosing which storage model to use thus often depends on how data will be used. Row-oriented storage models may be write-optimized and are commonly well suited for transactional queries (i.e. online transaction processing - OLTP), while column-oriented storage models may be read-optimized and are generally well-suited for analytical queries (i.e. online analytical processing - OLAP). Accordingly, conventional query processing schemes are tightly bound to the underlying storage model of the database being queried [5].

The term "record" may refer to a stored row (i.e. tuple) when it's referred to in a row store table. But when used in connection with a column store table, "record" may refer to a stored column (i.e. a column of a relation or entries described by an attribute of the relation).
The record may consist of multiple cells. The contents of a cell are generally referred to as an entry. An entry may be a value (e.g. the string "Johnson") or a value identifier (e.g. the integer "5"). Each record in a row store table and/or a column store table may be identified by a corresponding record identifier.
A column-oriented database management system (DBMS) is responsible for processing data tables stored as columns; it's also called a column engine. And a row-oriented database management system (DBMS) is responsible for processing data tables stored as rows; it's also called a row engine. A database may include a plurality of database management systems. The column engine and the row engine may be part of the same DBMS [5].

For example, a database having data stored in a row-formatted storage model may receive a mixed set of queries requiring transactional and analytical processing of that data. So, it may be desirable to process such queries without changing the format of the database storage model of the data being processed (i.e. without changing row store data to column store data and without changing column store data to row store data).

However, the current join algorithms are only suitable for databases with row storage model or column storage model but none of these algorithms supports join between tables with a mixed storage model (row and column) in cloud environment.

The authors in [12] described an approach for performing a mixed join indirectly in a conversion-based way. According to this approach, row table data is converted into column format and then the join is performed in the column engine, or column table data is converted into row format and then the join is performed in the row engine. However, conversion overhead for mixed join is usually not trivial from the perspective of performance and memory consumption. Therefore, it may not be desirable to use conversion-based mixed join queries in situations involving performance critical workload such as cloud environment.

IV. The Proposed Framework for Support Mixed Join Queries

Considering the above problems of the conversion-based mixed join algorithm, and given to deal with the huge volume of data in the cloud environment, the impact of these problems will be increased. This paper proposes a framework with a novel Conversion-free mixed join algorithm to handle such mixed join queries on relational data in the cloud data stores without the conversion overhead problems.
Considering the above problems of the conversion-based mixed join algorithm, and given to deal with the huge volume of data in the cloud environment, the impact of these problems will be increased. This paper proposes a framework with a novel Conversion-free mixed join algorithm to handle such mixed join queries on relational data in the cloud data stores without the conversion overhead problems. The cloud computing platform (a cluster) consisting of hundreds or thousands of PCs is responsible for data computing and storage. As Fig [4] shows, there are two types of nodes in the cluster: master nodes and slave nodes [11].
Master nodes store some meta data about the whole cluster including file name, file path, file replications, and so on, while slave nodes ,also called data nodes; store the regular data. The slave nodes store data records and their replicas.
The query on the cloud platform is different from central or parallel database. The data is partitioned and replicated across the several slave nodes in the system, allowing easy node additions or removals. Also, in the cloud platform, client query is often presented against the master nodes. After that the mater nodes decide which slave nodes are relevant to the query and then the query is passed to the slave nodes to do the query processing directly.
So the query in the cloud computing platform can be divided into two phases: locate the slave nodes which store the relevant data and process query on the slave nodes directly.
As the slave nodes are responsible for query processing, it uses a Mixed DBMS (MDBMS). The MDBMS is the storage manager component in charge of managing the data. MDBMS supports processing of both row oriented and column oriented tables. It supports the execution of the native mixed join queries on relational data stored on the cloud data stores.
The focus here is in the second phase to show in detail how the mixed join queries are processed without any conversion in table's orientation, and the first is similar to any cloud system.

© 2014, IJARCSSE All Rights Reserved

The column and row tables may be processed in a native state, i.e. conversion of row tables to column store format or of column tables to row store format is unnecessary because queries are processed using knowledge of the structures of the column store table and the row store table to find entries in the column store table matching entries in the row store table. The mixed join algorithm between database column and row tables is aware of both row store tables and column store tables, and is directly executable upon the data in its native form without requiring conversion between orientations.

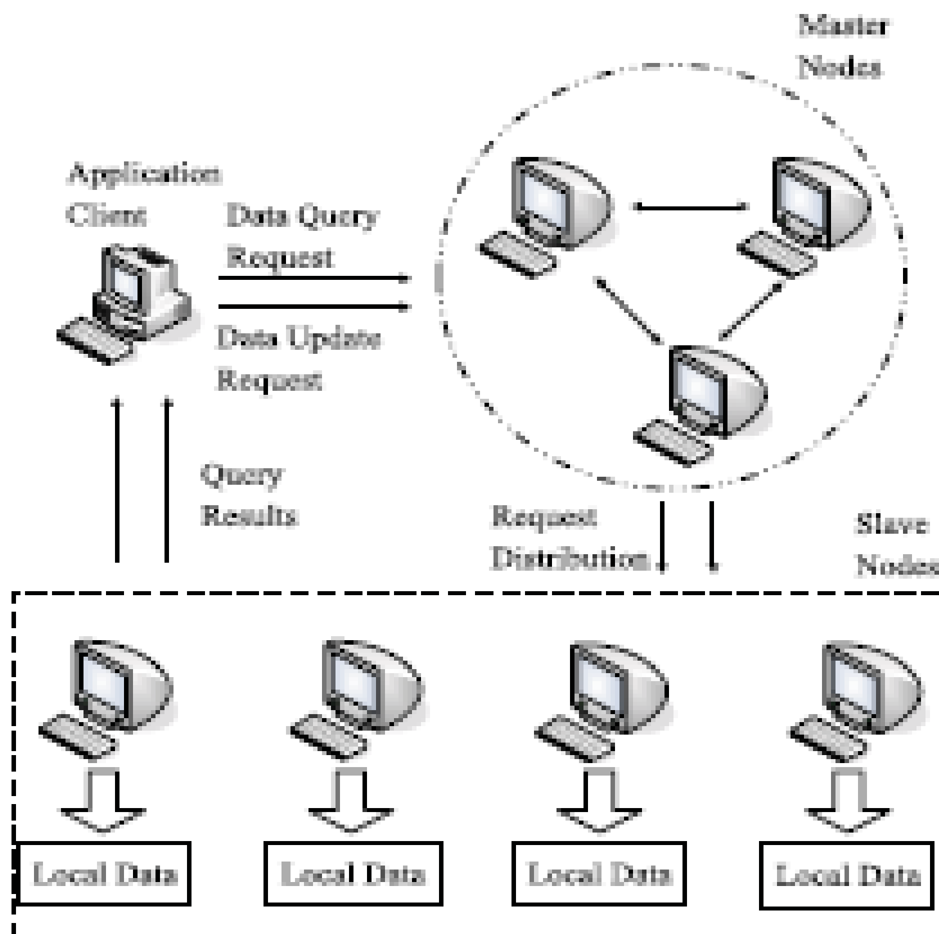


Fig [4] The Framework for query processing in the cloud

A column store table may have a corresponding column dictionary for each column. This dictionary maps the value identifiers appearing in the column to the values identified by the value identifiers. Optionally, the dictionary may have an identifier map (also referred to as an inverted index) to search for a column store record index (docid or doc_id) with given value identifier (vid). The column store record index may be implemented as a number that identifies the position of a column store entry in a column store record relative to the other entries in the column store record.

The main idea of the native mixed join algorithm is exploiting the column dictionary and the identifier map for efficient join processing. In particular, the native mixed join algorithm looks in a column dictionary for values matching a join condition, and returns matched records in a pipelined manner. So matched records may be searched for and returned in parallel. In other words, for each value in the row store join column, a search for the value in the column dictionary of the column store join column is performed in order to find the value identifiers corresponding to that value which are stored in the column store table. Thus, the native mixed join can take place utilizing row and column store data in its native form, by directly accessing values from both row and column store and returning matching pairs of rs_rid and col_rid.

V. The Mixed - Join algorithm

The following are the details of the mixed join algorithm, more particularly, a foreign key mixed join (1-to-N) algorithm, in the case where an identifier map is available. This algorithm is a type of inner join. In "1-to-N" the "1" refers to 1 record from a row store table and the "N" refers to an arbitrary number of rows of a column store table. Thus, a 1-to-N join may attempt to match each row of the row store table with all the rows of the column store table. For the foreign key (FK) mixed join, it can be assumed that join columns of the row table do not have duplicate values. Therefore, a column dictionary of a join column from the column store table can be accessed for each value in fetched row table records without a performance disadvantage.

© 2014, IJARCSSE All Rights Reserved

Fig [5] is a simplified chart listing steps involved in performing the mixed join for a query on relational data as shown in [5].

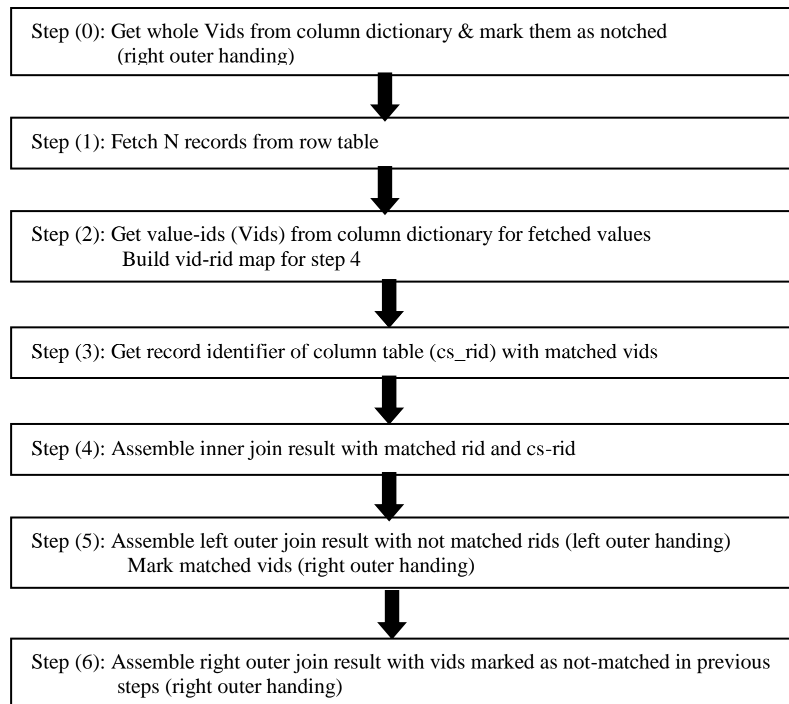


Fig [5] the steps of the mixed join query

The row oriented Department table is shown in Table [1], the column oriented Employee table is shown in Table [2], and the column dictionary is shown in Table [3]. In order to further explain the mixed join algorithm, consider following join query:

```
SELECT e.ename, d.location
FROM department d, employee e
WHERE d.dname = e.dname
```

The WHERE clause in the query specifies a join condition; the first join column of the row store table, and a second join column of the column store table, where the second join column references a first column store record of the column store table.

rid	dname	location
0x00	Sales	#101
0x04	Engineering	#103
0x08	Clerical	#204
0x0c	Marketing	#306
0x10	………	…………..

Table [1] Department (row table)

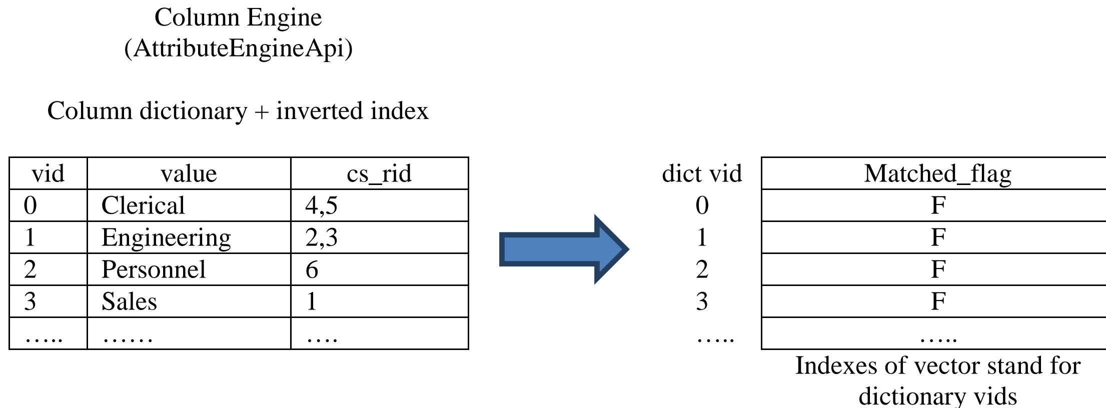
Cs_rid	Ename	dname
1	Rafferty	Sales
2	Jones	Engineering
3	Steinberg	Engineering
4	Robinson	Clerical
5	Smith	Clerical
6	John	personnel
	……….	……….

Table [2] Employee (column table)

Vid	Value	Cs_rid
0	Clerical	4,5
1	Engineering	2,3
2	Personnel	6
3	Sales	1
………..	…………	………..

Table [3] The Column dictionary + The inverted index

In the preliminary step (step 0) the value ids of the column dictionary (vids) are retrieved using the column engine. An output of this



vid	value	cs_rid
0	Clerical	4,5
1	Engineering	2,3
2	Personnel	6
3	Sales	1
…..	……	….

dict vid	Matched_flag
0	F
1	F
2	F
3	F
…..	…..

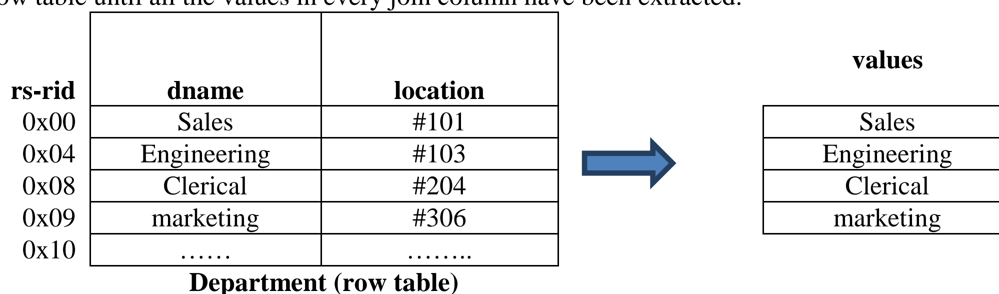
Fig[6] Step (0): Get whole Vids from column dictionary & mark them as not matched (right outer handing)

preliminary step is a matched_flag vector shown in Fig [6]. It's an array or a list of Boolean values whose indexes stand for dictionary vids.

In the first step (step1) of the FK mixed join algorithm, N records are fetched (i.e. extracted or copied) from the row table, (N refers to an arbitrary number of records).

The N records include the first join column in the row table.

A set of values in the first join column of the N records may be extracted, e.g. to form an array of string values. The set of values may include a first row store value and a second row store value. The first and second row store values may be in a column of the row store table referenced or specified by the first join column. Further sets of values may be extracted from the row table until all the values in every join column have been extracted.



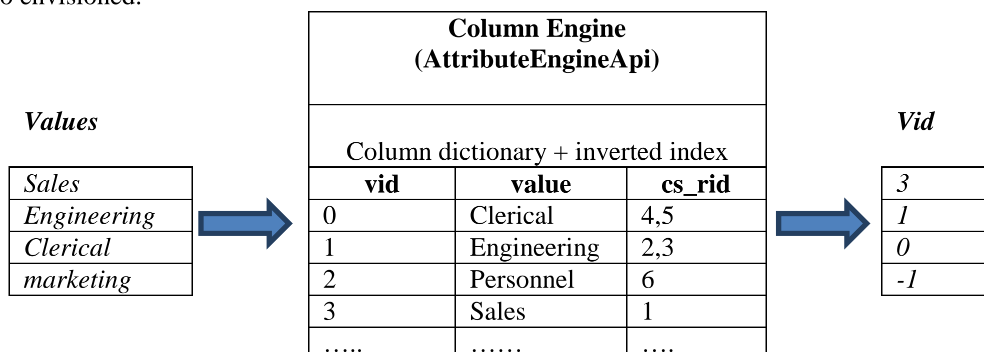
rs-rid	dname	location
0x00	Sales	#101
0x04	Engineering	#103
0x08	Clerical	#204
0x09	marketing	#306
0x10	……	……..

values
Sales
Engineering
Clerical
marketing

Fig [7] Step (1): Fetch N records from row table

As shown in Fig [7], the records are fetched (i.e. extracted from the row table and may be filtered according to the join condition (e.g. only values in the join columns of the records remain after filtering), resulting in filtered records. A string values array may be constructed from the filtered records.

In (step 2) shown in Fig [8], the string values array is input to the column engine, and the column dictionary is accessed to get the corresponding column store value identifier (vid), which is an integer in this example. The integer vids array of column store value identifiers is output. The vids array may include values indicating that there is a value in the string values array with no matching value in the column dictionary. Here, a vid of -1 indicates that there is no matching value in column dictionary. -1 is an example of an invalid column entry (i.e. an invalid value identifier). Other invalid column entries are also envisioned.



Values
Sales
Engineering
Clerical
marketing

vid	value	cs_rid
0	Clerical	4,5
1	Engineering	2,3
2	Personnel	6
3	Sales	1
…..	……	….

Vid
3
1
0
-1

Fig [8] step(2): Get value-ids (Vids) from column dictionary for fetched values

© 2014, IJARCSSE All Rights Reserved

Column Engine
(AttributeEngineApi)

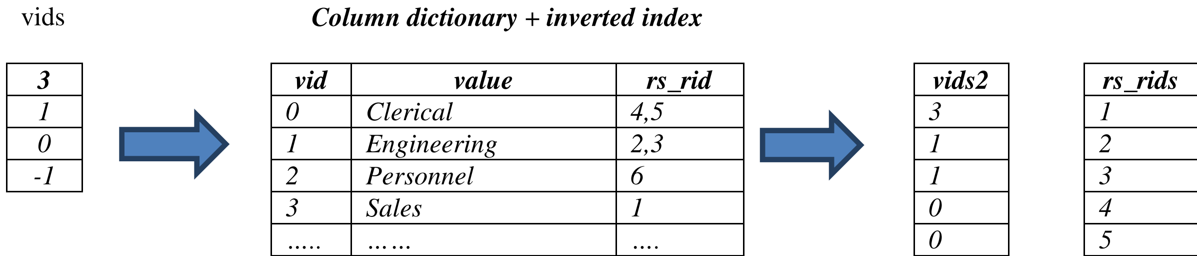


Fig [10] Step(3): Get record identifier of column table (cs_rid) with matched vids

Step 2 also involves constructing an intermediate map. This map is referred to as a rip4vid map. To construct this map, the vids array is matched with a corresponding rs_rid from the row table. The rid4vid map is built, to be used in finding the rs_rid based upon the vid input. Thus, the intermediate map (i.e. the rid4vid map shown in Fig [9]) may map column store value identifiers (vids) to row store row identifiers (rs_rids).

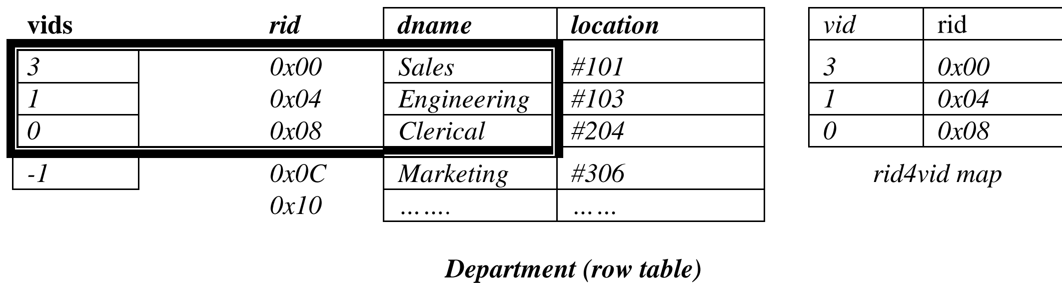


Fig [9] Step(2): Build vid-rid map for step 4

In the next step (step 3) shown in Fig [10], the corresponding doc_ids array is created. Specifically, the integer vids array is input to the column engine to search an identifier map to obtain the column store record identifier (doc_id). The column store record indexes (integer doc_ids array in this example) along with a corresponding set of value identifiers (vids2 array in this example) may be output.

In the next step (step 4) shown in Fig [11], the inner join result is assembled. This involves scanning the vids2 array having the corresponding doc_id. The rid4vid map is accessed to find the rs_rid. Then the rs_rid is matched with the doc_ids to form an inner join result. The inner join result may then be used to access values of the column store record referenced by the docids in order to obtain values of the column store record matching the join condition. The matching values may be materialized in a projected column (i.e. a materialized column). An identifier of a row of the projected column (col_rid) may be matched with a corresponding row store row identifier (rs_rid). Each rs_rid from the row store table may be matched with a row of a projected column from the column store table according to the join condition in order to form the result table. Each projected column may include a subset of the values in a column store record; in particular, values of the column store record matching the join condition. The projected column may comprise entries from a column store record; it may include entries from the column store record that match values from the row store record according to the join condition.

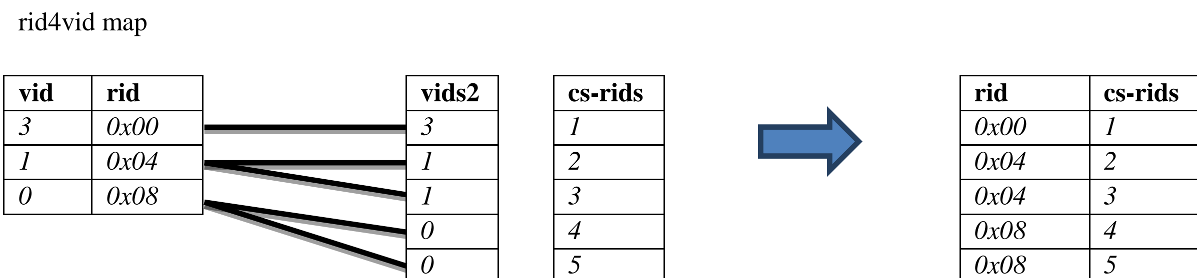


Fig [11] Step (4): Assemble inner join result with matched rid and cs-rid

© 2014, IJARCSSE All Rights Reserved

The next step (step 5) comprises outer join handling as shown in Fig [12]. If a left outer join is specified in the query, then the vids array is compared to the row table to return the rs_rid whose matching vid is -1, indicating null column value. -1 is an example of an invalid column entry. If a right outer join is specified in the query, then a matched_flag vector (similar to the matched flag vector) may be created and vids matching the join condition (e.g. vids with an integer value other than "-1") may be identified in the matched_flag vector. Value identifiers (vids) matching the join condition may also be referred to as valid vids. Value identifiers not matching the join condition may be referred to as invalid vids.

In the final step (step 6), when the right outer join is specified, the vids mapped to a matched_flag boolean value indicating false ("F") are identified in the matched_flag vector. Column store row identifiers in an index without a matched vid, are searched with the column engine. doc_ids that are not matched are returned with null row table value. as shown in Fig [13] vid 2 is the unmatched vid and doc_id 6 is found as unmatched (not matched) doc_id by the column engine.

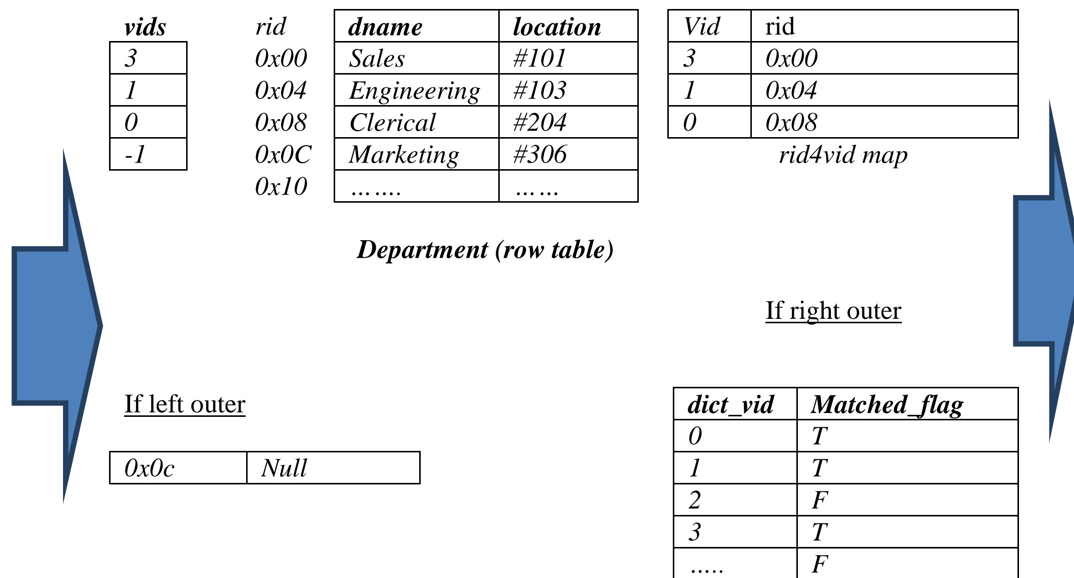


Fig [12] Step (5): Assemble left outer join result with not matched rids (left outer handing) or Mark matched vids (right outer handing)

Column Engine (AttributeEngineApi)

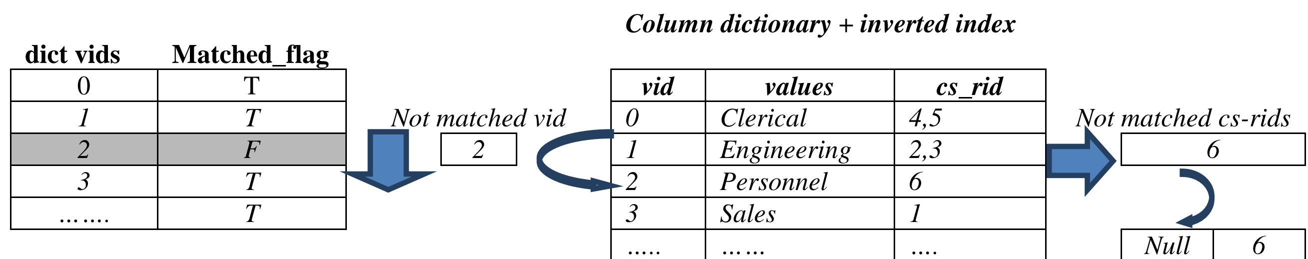


Fig [13] Step (6): Assemble right outer join result with vids marked as not-matched in previous steps (right outer handing)

For example, in the context of the right outer join, the first join column may reference a column of the row store table and the column store value identifiers match a value in the column of the row store table referenced by the first join column. Continuing the example, the boolean value of a value identifier is mapped to a value of true ("T") in the matched_flag when the column store value mapped to the value identifier in the column dictionary matches a row store value according to the join condition; accordingly, the boolean value of the value identifier is mapped to a value of false ("F") in the matched_flag when there is no row store value that matches (according to the join condition) the column store value mapped to the value identifier in the column dictionary.
The final result of a mixed join operator is an array comprising pairs of rs_rid (referring to the row table) and col_rid (referring to values from the column table).

© 2014, IJARCSSE All Rights Reserved

VI. CONCLUSION

This paper presents a framework for supporting native mixed join queries on relational data in cloud data stores. The native mixed join query algorithm performs joining between two tables with different orientations; column and row, without performing any conversion between the tables' orientations. This means that the join operation is executed upon the tables in its native orientation. The main goal of this framework is to optimize the mixed join queries processing in the cloud data stores by removing the overhead associated with tables' orientations conversions, especially that the conversion overhead for mixed join in the cloud is usually not trivial from the perspective of performance and memory consumption. Future work includes experimental evaluation of the proposed framework, and a performance comparison of the native and conversion based mixed join algorithms in the cloud data stores. These experiments must assess the extent of improvement or optimization in processing of mixed join queries on relational data in cloud data stores.

REFERENCES

[1] Abadi, Daniel J. *Query execution in column-oriented database systems*. Diss. Massachusetts Institute of Technology, 2008.

[2] NANDA, Ruchi. Performance enhancement techniques of cloud database queries. 2013.

[3] ABADI, Daniel J.; MADDEN, Samuel R.; HACHEM, Nabil. Column-Stores vs. Row-Stores: How different are they really?. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008. p. 967-980.

[4] LI, Jinsheng. *Cloud database query*. 2013. PhD Thesis. California State University.

[5] YOON, Yongsik; JEONG, Chanho; CHA, Sang Kyun. *Mixed Join of Row and Column Database Tables in Native Orientation*. U.S. Patent Application 2013/0151502A1, Jun. 13, 2013.

[6] JAYASHREE, J.; RANICHANDRA, C. JOIN ALGORITHM FOR EFFICIENT QUERY PROCESSING FOR LARGE DATASETS. *Asian Journal of Computer Science & Information Technology*, 2012, 2.3.

[7] HAN, Xixian; LI, Jianzhong; YANG, Donghua. PI-Join: Efficiently processing join queries on massive data. *Knowledge and information systems*, 2012, 32.3: 527-557.

[8] DONKENA, Kaushik; GANNAMANI, Subbarayudu. Performance Evaluation of Cloud Database and Traditional Database in terms of Response Time while Retrieving the Data. *Electrical Engineering*, 2012.

[9] WANG, Jinbao, et al. Indexing multi-dimensional data in a cloud system. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010. p. 591-602.

[10] ARORA, Indu; GUPTA, Anu. Opportunities, Concerns and Challenges in the Adoption of Cloud Storage. *International Journal of Computer Science and Information Technologies (IJCSIT)*, 2012, 3.3: 4543-4548.

[11] WU, Jiyi, et al. Cloud storage as the infrastructure of cloud computing. In: *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*. IEEE, 2010. p. 380-383.

[12] JEONG, Chanho, et al. *Processing Database Queries Using Format Conversion*. U.S. Patent Application 12/982,673, 2010.

[13] EGGER, Daniel. *SQL in the Cloud*. 2009. PhD Thesis. Master Thesis ETH Zurich, 2009.

[14] MICHEL, Daniel. Databases in the Cloud. *Doktorarbeit, HSR University of Applied Science Rapperswil*, 2010.

[15] WU, Jiyi, et al. Cloud storage as the infrastructure of cloud computing. In: *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*. IEEE, 2010. p. 380-383.

[16] KRANEN, Philipp, et al. The ClusTree: indexing micro-clusters for anytime stream mining. *Knowledge and information systems*, 2011, 29.2: 249-272.

[17] SHATDAL, Ambuj; KANT, Chander; NAUGHTON, Jeffrey F. *Cache conscious algorithms for relational query processing*. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[18] TAN, Kian-Lee, et al. Efficient join processing using partial precomputation. *Knowledge and Information Systems*, 1999, 1.4: 481-514.

[19] HE, Bingsheng; LUO, Qiong. Cache-oblivious databases: Limitations and opportunities. *ACM Transactions on Database Systems (TODS)*, 2008, 33.2: 8.

[20] CHEN, Shimin, et al. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 2007, 32.3: 17.

[21] OMahonys, Conor. "What You Need to Know about Column-Oriented Database Systems." *Conor OMahonys Database Diary*. N.p., n.d. Web. 10 Mar. 2014.

[22] Ghosh, Abhishek. "Cloud Database as a Service : Why It Is Effective Solution." *The Customize Windows*. N.p., n.d. Web. 12 Mar. 2014.

[23] Calpont Corporation. "Why Should I Check Out a Column Database?" N.p., n.d. Web. 5 Mar. 2014.

[24] ARORA, Indu; GUPTA, Anu. Opportunities, Concerns and Challenges in the Adoption of Cloud Storage. *International Journal of Computer Science and Information Technologies (IJCSIT)*, 2012, 3.3: 4543-4548.

[25] JONES, M. Tim. Anatomy of a cloud storage infrastructure. *IBM developer works (November 30, 2010)*, 2010.

[26] Berchtold, Stefan, Daniel A. Keim, and Hans-Peter Kriegel. "The X-tree: An index structure for high-dimensional data." *Readings in multimedia computing and networking* (2001): 451.

[27] CHEN, Hanxiong, et al. Indexing expensive functions for efficient multi-dimensional similarity search. *Knowledge and information systems*, 2011, 27.2: 165-192.

[28] "Wikipedia. Join (SQL)." *Wikipedia. Join (SQL).* N.p., n.d. Web. 30 Mar. 2014.

© 2014, IJARCSSE All Rights Reserved

[29] RYENG, Norvald. Improving Query Processing Performance in Large Distributed Database Management Systems. 2011.

[30] WEI, Zhou; PIERRE, Guillaume; CHI, Chi-Hung. *Consistent join queries in cloud data stores*. Tech. Rep. IR-CS-068, Vrije Universiteit, Amsterdam, The Netherlands, 2011.

[31] ZHANG, Guigang, et al. Massive Data Query Optimization on Large Clusters. *Journal of Computational Information Systems*, 2012, 8.8: 3191-3198.

[32] KURUNJI, Swathi; GE, Tingjian; CHEN, Cindy X. Multi-Join Query Optimization for Read-Optimized Data Warehouse in a Cloud Environment.

[33] WEI, Zhou; PIERRE, Guillaume; CHI, Chi-Hung. Scalable join queries in cloud data stores. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012. p. 547-555.

[34] BLOOR, Robin. WHAT IS ACloud DATABASE?. 2011.

[35] VALLURI, Satyanarayana R., et al. *Methods, systems, and computer-readable media for providing a query layer for cloud databases*. U.S. Patent Application 13/304,043, 2011.

[36] *MEHTA, Navneet. A REVIEW MODEL ON QUERY OPTIMIZATION USING OPEN SOURCE DATABASE. International Journal of Scientific & Engineering Research, Volume 4, Issue 4, April-2013 ISSN 2229-5518.*

[37] METH, Halli; DEKHTYAR, Alexander. The History of Join Query Support for the Cloud.

[38] SILVA, Yasin N.; LARSON, P.-A.; ZHOU, Jingren. Exploiting common subexpressions for cloud query processing. In: *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 2012. p. 1337-1348.

[39] MA, Ding, et al. *Multi-join database query*. U.S. Patent Application 13/349,366, 2012.

© 2014, IJARCSSE All Rights Reserved