Optimizing Communication for Multi-Join Query Processing in Cloud Data Warehouses

Swathi Kurunji, Tingjian Ge, Xinwen Fu, Benyuan Liu, Cindy X. Chen
Computer Science Department, University of Massachusetts Lowell
Lowell, Massachusetts, USA
Email: {skurunji, ge, xinwenfu, bliu, cchen} @ cs.uml.edu

ABSTRACT

In this paper, we present storage structures, PK-map and Tuple-index-map, to improve the performance of query execution and inter-node communication in Cloud Data Warehouses. Cloud Data Warehouses require Read-Optimized databases because large amount of historical data are integrated on a regular basis to facilitate analytical applications for report generation, future analysis, and decision-making. This frequent data integration can grow the data size rapidly and hence there is a need to allocate resource dynamically on demand. As resource is scaled-out in the cloud environment, the number of nodes involved in the execution of a query increases. This in turn increases the number of inter-node communications. In queries, join operation between two different tables are most common. To perform the join operation of a query in the cloud environment, data need to be transferred among different nodes. This becomes critical when we have huge amount of data (in Terabytes or Petabytes) stored across a large number of nodes. With the increase in number of nodes and amount of data, the size of the communication messages also increases, resulting in even increased bandwidth usage and performance degradation. In this paper, we show through extensive experiments using PlanetLab Cloud that our proposed storage structures PK-map and Tuple-index-map, and query execution algorithms improve the performance of join queries, decrease inter-node communication and workload in Cloud Data Warehouses.

Keywords: Communication Cost, Read-Optimized Database, Data Warehouse, Cloud Storage, Query Optimization, Multi-join Query

1. INTRODUCTION

Historical Data from one or more heterogeneous data sources are extracted using ETL (Extract Transform and Load) technique and stored in a central repository called Data Warehouse or Data Marts (smaller version of Data warehouse), so that the data can be easily accessed for analysis. Data Warehouse is an organized collection of databases containing such historical data, which may represent the business history of an organization, or biological data of diseases, or agricultural data. Analysts use this data for online analysis and report generation, which require quick responses to iterative complex analytical queries. There are several good commercial
OLAP (Online Application Processing) applications for data warehouse analysis such as EMC’s Greenplum, IBM’s InfoSphere, Microstrategy, OracleBI, SqlServer, and Vertica.

In the past several decades, read-optimized databases have gained popularity in read intensive analytical workloads such as data warehousing and business intelligence applications. In these databases, data are partitioned and organized in many different ways such that it can be accessed quickly. Organization of data may be row-oriented, column-oriented or hybrid, depending on the type of application, usage or queries. Row-oriented databases store rows of a table sequentially on physical storage, whereas column-oriented databases store attribute values of one or more column (called projection) sequentially on physical storage [11] [15]. To achieve high level of parallelism, these tables or projections can be further horizontally divided and distributed over multiple nodes/systems in a network.

Cloud computing and storage has gained attention from researchers and consumers in recent years. It is designed to provide dynamically scalable resources to consumers, eliminating the hassle of investment and maintenance. Many commercial products from Amazon, Microsoft, EMC and IBM provide cost effective solutions like hourly or monthly or yearly billing [2]. Cloud provides an environment where the end user can perform tasks as if the data is stored locally when it is actually stored in remote systems. Providing such an environment needs powerful computing, fast execution strategies for tasks and high-speed communication network.

In the cloud architecture, depending on the availability of resources such as nature of application, storage space, and CPU cycles, data is distributed to different nodes. In addition, the physical location of the data may dynamically change from one node to another.

In traditional distributed databases, to reduce the inter-node communication during the execution of a query, tables are horizontally partitioned on the join attributes and related partitions are stored on the same physical system. In cloud environment, it is not possible to ensure that these related partitions are always stored on the same physical system. Thus, execution of queries in cloud data warehouses becomes more complicated when queries contain multiple joins between partitions of different tables stored in different nodes. These joins need back and forth communication among the query execution nodes to find the correct result. This heavy communication among the nodes will have adverse affects on the performance of the query and increase network traffic.

A long stream of research work has been done to improve the join operation in distributed read-optimized databases and network attached storages. Some of them are [4] [7] [8] [9] [10] [12] [14] [15] [16].

Efficient ways to concurrently execute multiple queries in order to increase the throughput is discussed in [9] and [14]. In [15], authors present vertical storage structure and indexing to handle record-oriented query. This vertical storage structure has improved the performance of data warehouse applications in the order of magnitude and has gained lot of attention. Paper [6] compares the actual difference between column and row-oriented DBMS in both physical and logical levels. Authors of [5] provide information on the internals of the column-oriented DBMS and the open research problems in the context of column-store systems, including physical database design, indexing techniques, parallel query execution, replication, and load balancing.

Authors of [7] [8] present query processing in a distributed environment using central master nodes which coordinate the mapping of local result to the global result, and communicate the global result to the query execution nodes. Efficient data access strategies and data distribution strategies to improve the performance of the query is discussed in [12] [16]. Paper [4] compares different architectures and proposes that the shared nothing architecture is well suited for cloud environment.

To the best of our knowledge, not much has been done to increase the independence of nodes executing multi-join query and decrease the communication between those nodes in a cloud environment. The scale-out and scale-in of resource in the cloud environment should not increase the network bandwidth usage for multi-join query execution. To ensure this we need to
reduce communications between the nodes (i.e., by providing a strategy which allows nodes to execute queries as independently as possible) and reduce the size of the partial query results exchanged in each communication. Thus in this paper, we take a different perspective and concentrate on optimizing the multi-join query execution by reducing the communication between nodes in cloud environment.

We take advantage of relationship between tables in the Data Warehouse Database and design two storage structures, which keeps information about those related tables. Our proposed storage structure stores information about the primary and foreign keys of tables. We create our first storage structure called PK-map (PrimaryKey-map), for every primary key of the tables, which is referenced as foreign key in other tables. PK-map stores the information of row mapping (record mapping) between the primary key of dimension table and our proposed second storage structure. Our second storage structure, Tuple-index-map stores information of row mapping (record mapping) between the PK-map and the foreign keys of the fact tables. We also store some required header information in our maps. These maps are then horizontally partitioned based on the primary key of the dimension table that are stored in the nodes and distributed to it.

With this information stored in maps each node can execute query as independently as possible by reducing the communication between the nodes during query execution. So, each node communicates only when there is a join between two tables with the predicate on the non-primary or non-foreign key column.

We store record-ids of the foreign keys rather than the actual attribute values in our maps. When there is a requirement of communication between the nodes we send record-ids (or PK-map indices) instead of attribute values. This reduces the size of the intermediate result exchanged and thus the communication overhead. Hence, query processing using PK-maps not only improves the performance of each query but also helps to reduce the network traffic.

A preliminary version of this paper appeared in CloudCom 2012 [13], where we presented the basic idea of the proposed storage structures and performance evaluation on small network of 4 virtual nodes. In this paper, we make the following additional contributions: First, we provide a more detailed description of our proposed structures, query processing algorithm and its advantages such as data distribution and data skew handling. Second, provide extensive performance study on large-scale cloud called PlanetLab to demonstrate the effectiveness of the proposed approach.

The remainder of this paper is organized as follows: Section 2 provides information on TPC-H Star Schema, which we use in the rest of the paper for analysis and performance study. Section 3 explains proposed storage structures, PK-map and Tuple-index-map. Section 4 provides an algorithm to process multi-join query. Section 5 analyzes the storage space of PK-map and Tuple-index-maps, and the size of intermediate result communication, Section 6 explains about data distribution, manipulation and skew handling, Section 7 shows the performance evaluation of small-scale as well as large-scale cloud network. Next Section states acknowledgements, conclusion and references.

2. TPC-H BENCHMARK

Star and Snowflake Schema representations are commonly used in read-optimized Data Warehouses. In the rest of this paper we use the star schema from "TPC BENCHMARK H Standard Specification Revision 2.15.0" (Fig. 1).
Transaction Processing Performance Council (TPC) is a non-profit organization, which defines transaction processing and database benchmarks, to provide objective, verifiable performance data to the community. TPC produces benchmarks that measure performance in terms of how many transactions a given system and database can perform per unit of time. TPC-H is a decision support benchmark, which is designed to evaluate the functionalities of business analysis applications (Online analytical processing applications). Fig. 1 is a schema of an industry, which must manage, sell and distribute its products worldwide.

3. PK-MAP AND TUPLE-INDEX-MAP STRUCTURES

3.1 PK-MAP STRUCTURE

Data from two different tables are normally accesses by performing join operation on primary and foreign key attributes of those tables. Then, the required attribute values of those two tables are filtered. We create a PK-map (i.e., PrimaryKey map) for each of those primary keys in the tables of the star schema. A PK-map will have one column for primary key and one column for each of the foreign keys referencing this primary key as shown in Table 1 and Table 2. In the primary key column we store the primary key attribute values and in the foreign key column we store the logical record-id of the Tuple-index-map of the foreign key. This logical record id runs from 1 to n. The PK-maps are sorted on primary key values, which allow us to apply run length encoding on foreign key logical record-ids and reduce the size of the map to a great extent. Thus, the overall size of the map will be proportional to the number of records in the dimension table (Size of these maps is discussed in Section 5.1).
Dimension tables are usually smaller in size, which stores the descriptive data such as region or nation table in Fig. 1. On the other hand, fact tables are very large, which stores data of the transactions such as lineitem table in Fig. 1.

In Table 1 and Table 2, first column stores the primary key value and subsequent columns store logical record-id (which is the starting record-id) of the Tuple-index-map.

### 3.2 TUPLE-INDEX-MAP STRUCTURE

If the foreign key table is sorted on the foreign key attribute value, then we do not require Tuple-index-map for that relationship. In this case, logical record-id of the PK-map will be the actual record-id of the foreign key table. In Fig. 1, as PART and PARTSUPP tables are sorted on “partkey” attribute, PartKey-map does not require Tuple-index-map. Similarly in column-oriented databases, if even one projection of the table is sorted on foreign key attribute, then the logical record-id of the PK-map will be the actual foreign key record-id.

For all those tables, which are not sorted on foreign key attribute value, we create a tuple-index-map as shown in Table 3. This Tuple-index-map will store the mapping between the logical and actual record-id of the foreign keys in the foreign key table.

Both PK-maps and Tuple-index-maps are horizontally partitioned by primary key attribute and distributed to all the nodes containing the corresponding data. Replication of these maps can be done to improve the throughput or disaster recovery. For example, Table 1 is partitioned into 3 partitions where rows 0 and 1 are assigned to partition 1, rows 2 and 3 to partition 2 and row 4 to partition 3. Accordingly Tuple-index-map of foreign keys is partitioned to correspond to the partitions in Table 1. Tuple-index-map is partitioned into rows 0 to 9, 10 to 19 and 20 to 24. Similarly, all the other PK-maps and Tuple-index-maps are horizontally partitioned.

In the schema of Fig. 1, nation and region tables are very small with only 5 and 25 rows and require less storage space. So, it is not necessary to partition these tables. Instead, these tables can be replicated among all the nodes to improve the performance. But, this may not be the case always. There will be large dimension tables like customer table, which grows dynamically as the new customers are added. To show the scenario where the dimension tables are large and need to be partitioned, we have partitioned the region and nation table.

With this information on PK-maps and Tuple-index-maps each node can execute query independently even when there is a join between two different tables that are located in two different nodes. For example, consider a query: Find all suppliers of region “EUROPE”. To find the suppliers we need to join supplier, nation and region tables of Fig. 1. If these tables are stored
among different nodes then we have to communicate with those nodes to get the mapping between the tables. But with our map structures, we can look up for the mapping and process the query. This reduces the communication among nodes during query execution. In this approach, scanning maps performs join between different tables. We will show the detailed processing of some of the TPC-H queries in the coming sections.

With the TPC-H schema of Fig. 1, we need to create 7 PK-maps RegionKey-map, NationKey-map, SuppKey-map, PartKey-map, PartsuppKey-map, CustKey-map and OrderKey-map. Structure of RegionKey-map and NationKey-map are shown in Table 1 and Table 2. Table 1 contains the mapping between region keys of region and nation tables. Table 2 contains the mapping between nation keys of nation, supplier and customer tables (Fig. 1). Similarly we can create the remaining 5 maps. Table 3 gives the mapping between logical and actual record-id of the nation table, when the nation table is sorted on nation key attribute.

4. MULTI-JOIN QUERY PROCESSING

4.1 REFERENCE GRAPH

We created reference graph for TPC-H Schema (Fig. 1) as shown in Fig. 2. This graph is used to filter as many unwanted data as possible during the early stages of the query processing. In Fig.2, each rectangle box represents a table of the star schema and directed arrows connecting these boxes represent the relationship between the tables. i.e., arrow connecting region table to nation table means the primary key of region table is referred as foreign key in the nation table; “d” gives the depth of the table, i.e., number of other tables linked to it starting from the table with no reference (d=0).

![Figure 2: Reference graph](image)

4.2 QUERY PROCESSING ALGORITHM

We will use Algorithm1 to process multi-join query along with the help of proposed PK-maps, Tuple-index-maps and reference graph. Ad-hoc query and reference graph are provided as input to the algorithm. In this algorithm we first retrieve all the tables in the "from" clause of the query and sort them by the corresponding d value in the reference graph. Then, we process each table predicates starting from the first table of the above-sorted order.

For all those join predicates present in the query, we scan PK-map and Tuple-index-maps. Instead of communicating with peer nodes and then performing join operation. We store the result of applying predicates in the form (rec-id₁, rec-id₂, rec-id₃, ...), where rec-id₁ will be the record-id of first table with d=0 and rec-id₂ will be the record id of the second table which
matches the record rec-id1 and so on. While applying predicate on table i, if we do not find any matching record, then we eliminate the previously stored result (rec-id1, rec-id2, rec-id3, rec-id4, ...). After processing all the tables, the final result is constructed using the remaining mapping (rec-id1, rec-id2, rec-id3,...) to retrieve non-primary and foreign key attribute values of the tables.

**Algorithm 1** Query Processing Algorithm

```
Input: query Q, reference graph G
Output: Result of query Q
1. Let T be an array of tables referenced in Query
2. Sort T based on d value of G
   (If value is shown in Fig. 2)
3. for each table t ∈ T do
4.   if there is a predicate on non-PK/non-FK then
5.     if d == 0 for t then
6.       Apply predicate on t to get the record id's
7.       Store the record-id mapping in the format
8.       (rec-id1, rec-id2, ...).
9.   Communicate if necessary with other nodes
10. else if any table t1 with d1 ≤ d referenced by t then
11.   Apply predicate on t
12.   Update the mapping with rec-id's of t
13. Perform line 9
14. Eliminate mappings which has no match for t
15. else
16.   Perform similar to line 6, 9, 14
17. end if
18. else if there is a predicate on PK or FK then
19.   if d == 0 for t then
20.     Scan PK-map and tuple-index map
21.     Perform line 6 to 8
22. else
23.     Scan PK-map and tuple-index map for those rec-id's stored for table t1 with d1 ≤ d that is referenced by t
24.     Perform 12 and 14
25. end if
26. end if
27. end for
28. Scan tables of T for final mappings(rec-id1,...) to get the values of other attributes in the select statement of Q
29. return Result
```

### 4.3 CASE STUDY OF TPC-H QUERIES

In [13], we chose 3 out of 22 TPC-H queries for detailed analysis and performance study as shown in Fig. 3, Fig. 4 and Fig. 5. We have provided detailed explanation of Example 1 query processing using the proposed Algorithm 1. We have also showed how the proposed approach reduces the number of communications while performing join predicates in the query. Interested readers might want to have a look at the case study of [13] to get more understanding about the working of Algorithm 1.

Figure 3: TPC-H Query 2 (Source[3], p 30)

**Example 1**

```sql
select s_acctbal, s_name, n_name, p_partkey, p_partkey = p_partkey
and s_suppkey = p_suppkey
and p_size = 15
and p_type like '%BRASS'
and n_nationkey = n_nationkey
and r_regionkey = r_regionkey
and r_name = 'EUROPE'
and p_supplycost = (select min(p_supplycost)
from partsupp, supplier, nation, region
where p_partkey = p_partkey
and s_suppkey = p_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'EUROPE')
order by s_acctbal desc, n_name, s_name, p_partkey;
```
Similarly we can process Example o asi

Example 2

```
select n_name, 
    sum(l_extendedprice * (1 - l_discount)) as revenue
from customer, orders, lineitem, supplier, nation, region
where c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = n_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= date '1994-01-01'
    and o_orderdate < date '1994-01-01' + interval '1' year
group by n_name
order by revenue desc;
```

Example 3

```
select c_custkey, c_name, 
    sum(l_extendedprice*(1-l_discount)) as revenue,
    c_acctbal, n_name, c_address, c_phone, c_comment
from customer, orders, lineitem, nation
where c_custkey = c_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date '1993-10-01'
    and o_orderdate < date '1993-10-01' + interval '3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
group by c_custkey, c_name, c_acctbal, 
    c_phone, n_name, c_address, c_comment
order by revenue desc;
```

On the other hand, in the state-of-the-art approach each join between two tables requires a communication with other nodes. This is because a node does not know whether the other node has any record, which matches the current node’s record. So communication is needed between nodes before each join operation in the query.

5. ANALYSIS OF STORAGE AND COMMUNICATION MESSAGE SIZE

5.1 PK-MAP AND TUPLE-INDEX-MAP SIZE

As we discussed in Section 2, number of rows of a PK-map is equal to the number of rows of a dimension table (i.e., the primary key table). Number of columns of a PK-map will be equal to the number of foreign keys referencing the primary key plus one for the primary key column. Some header information like partition and node information will also be stored to locate the table data stored in different nodes. Thus, the overall size of each map will be,

\[
\text{Size Of PK-map} \leftarrow S_1 + \sum_{i=1}^{n} S_2 [i] + c \quad (1)
\]

- \(S_1\) (Size of 1st column) \(\leftarrow (\text{Num. of rows of PK}) \times (\text{Size of PK attribute})\)
- \(S_2\) (Size of other columns) \(\leftarrow (\text{Num. of rows of PK}) \times (\text{Size of Record-id})\)

i.e., \(S_2 \leftarrow (\text{Num. of rows of PK}) \times (\text{Size of Integer})\)

‘c’ is the size of the header information,
‘n’ is the number of foreign keys (FK) referencing primary key (PK)
‘\(S_2 [i]\)’ is the size of the foreign key column \(i\).

Number of rows of Tuple-index-map will be equal to the number of rows of the foreign key table. Hence, the size of the Tuple-index-map will be the size of integer times the number of rows of the foreign key table.

\[
\text{Size Of Tuple-index-map} = (\text{Num. of rows of FK}) \times (\text{Size of Integer}) \quad (2)
\]

For example, consider 5 regions and 25 nations (each region has 5 nations). PK-maps, RegionKey-map and NationKey-map will look like Table 1 and Table 2. Tuple-index-map: RegionKey-tuple-index-map will look like Table 3. Let us consider the size of region key be varchar(10) and nation key be varchar(10).

Table I size: \(S_1 = 50\) bytes, \(S_2 = 40\) bytes and Size of RegionKey-map is \(90 + c\) bytes
Table II size: $S_1 = 250$ bytes, $S_2 = 200$ bytes and Size of NationKey-map is, $650 + c$ bytes
Table III size: Num. of rows of nation table is 25. Size of Tuple-index-map is, $25 * 4$ bytes

Similarly we can calculate the size of the remaining maps and find the total storage space needed to store all the required maps.

Size of PK-map will always be small because the dimension table will always be smaller than the fact table and dimension table will grow slower than the fact table. PK-map will grow only when the new primary key is added to the dimension table. When a new row is added to fact table, we just have to update the value in the PK-map and append a row to the tuple-index-map. Tuple-index-map grows proportionally to the fact table.

5.2 INTER-NODE COMMUNICATION MESSAGE SIZE

Inter-node communication message size is the size of the intermediate query result sent from one node to another. In our approach, nodes exchange map indices with each other during the query processing. But in general distributed query execution, nodes exchange join attribute values. Join attribute values might not always be integer and can be a large string. Thus, the proposed approach reduces the size of each inter-node communication overhead.

For example, as shown in Section 4.3, using our approach nodes communicate only record-ids during the execution of Example 1 query. First, record-id of EUROPE region is communicated. So, the message size is $4 + c$ bytes, where 4 is the size of record-id (integer) and $c$ accounts for header information. In general approach the message size will be $(10*4)+c$ bytes, where 10 is the size of region key attribute (varchar). In this case the message size is small, but when results of large tables need to be exchanged the message size will be significantly larger.

We will show the total size of the intermediate results exchanged in the performance evaluation section.

6. DATA DISTRIBUTION, MANIPULATION AND SKEW HANDLING

6.1 DATA DISTRIBUTION AND SKEW

Data distribution is a process of partitioning and distributing dataset into one or more systems/nodes forming a cluster, so that each node can compute tasks in parallel to get better performance. It also contributes towards load balancing, reliability, availability and disaster recovery. But, it is important to carefully choose the partitioning strategy while distributing the dataset in order to achieve the maximum performance. Otherwise it will cause data skew.

Non-uniform distribution of dataset is called data skew. It has direct impact on distributed query execution leading to poor load balancing and high response time. As stated in the paper [19], data skew can be classified into intrinsic skew and partition skew. Intrinsic skew occurs due to the uneven distribution of attribute values in the dataset. For example, there are more customers from United States than United Kingdom. Partition skew is due to the uneven partition of data. Since join operation is the most expensive operation in the entire query execution, some distributed database systems partition data on join attribute and then distribute related partitions of tables to the same physical system. This is done to avoid redistribution of data to perform join operations. But, this kind of partitioning causes partition skew.

Our proposed map structures eliminate the above problem of partition skew caused by join attribute partitioning. Since we store the relationship information of the tables in our maps, we can distribute data randomly without considering the join attributes. In addition, we can easily partition the data to avoid intrinsic skew during the data upload. Since our maps are independent of data value distribution and depend on relationship between tables, we won’t be having any problem of data distribution.
6.2 DATA MANIPULATION

Data manipulation operations like insert, delete and update operations are done along with the batch upload in Cloud Data Warehouses. Data manipulation operations are less important in this scenario. Still we need to evaluate those operations in order to analyze whether there is any overhead, which makes it unusable.

Since PK-map and Tuple-index-map contains only primary key values and logical record-ids, we don’t have to update maps when there is an update operation on the data values other than primary key attribute. Usually update operation will occur on non-primary key or non-foreign key attributes. If there is update operation on primary or foreign key values, then we need to do same operations as for data tables (delete a row and insert new row).

For insert operation, if it is an insert on dimension table, we need to append new row at the end of the PK-map and append corresponding rows in the Tuple-index-map. If the insertion is on fact table, then we just increment the count of corresponding primary key value in the PK-map and add new row to the Tuple-index-map (Section 3).

For delete operation, if it is on dimension table, we need to delete a row from the PK-map and corresponding rows from Tuple-index-map. There is no need of sorting these maps. If the delete is on fact table, we decrement the count of corresponding primary key value in the PK-map and delete corresponding row of the Tuple-index-map.

7. PERFORMANCE EVALUATION

In this section, we present the performance study to evaluate the effectiveness of our proposed PK-map and Tuple-index-map structures using the Algorithm1. First, we will analyze the performance on small-scale network of 4 virtual nodes. Second, we will study the performance on large-scale cloud network called “PlanetLab” with 50 nodes and 150GB of data.

7.1 SMALL-SCALE NETWORK

Here, we measure the performance of average query execution time, number of communications made by nodes, total size of messages exchanged by the nodes, and the change in PK-map and Tuple-index-map size along with the change in data size.

Our test environment is a network of four virtual CentOS machines. Each of these machines has 2.6GHz Six-Core AMD Opteron Processor, 2GB RAM and 10GB hard disk space. Also, these machines are running Oracle 11g. All the map structures are loaded into the main memory before each experiment begins. Here we can take advantage of main memory databases if the data size is huge (in Petabyte or Exabyte) [20] [21] [22]

To perform experiments, we generated data using data generator, dbgen provided by TPC-H benchmark and distributed it to all four nodes. We generated 10GB of data for the experiments in the Sections 7.1.1 to 7.1.3 below. We generated PK-maps and tuple-index-maps, and horizontally partitioned them into four partitions and distributed them to the four nodes with corresponding data. We used same partition keys as data partition. We took same three queries, which we have analyzed in Section 4.3 to analyze the performance.

7.1.1 Average Query Execution Time

Comparison of average time taken by the queries in Fig. 3, Fig. 4 and Fig. 5 are shown in Fig. 6. In this graph (Fig. 6), queries are on the x-axis and time taken (in seconds) on the y-axis. On x-axis, for each query we have vertical 2 bars, where first bar (in red) shows the time taken by our approach and the second bar (in blue) shows the time taken with the general approach.

Each node has to communicate partial result or the join attribute values with its peers for every join predicate present in the query. Thus each node has to undergo additional message processing. Also, based on the data in Fig. 6, it is clear that, the query execution time required by the state-of-the-art approach is more than our approach.
7.1.2 Number of Inter-node Communications

As each inter-node communication has some inherent overhead, we also compare the number of messages in addition to the message size.

Comparisons on the total number of messages exchanged during the execution of queries are shown in Fig. 7. In this graph (Fig. 7), queries are on the x-axis and the number of messages is on the y-axis. Example 1 query execution involves nine communications (i.e., 3 times communication with 3 nodes) using our approach and fifteen communications in the state-of-the-art approach. Example 2 query execution involves six communications in our approach and fifteen communications in the state-of-the-art approach. Example 3 query execution involves three communications with our approach and nine communications in the state-of-the-art approach.

Based on the result in Fig. 10, we can infer that the number of communications in our approach is fewer than in the general approach. This optimization has a big impact, since the number of communications increases rapidly with increase in number of nodes.

7.1.3 Size of Inter-node Communication Messages

To compare the inter-node communication message size between the proposed and the state-of-the-art approach, we added up the size of all messages exchanged while executing the queries. The comparison results are shown in Fig. 8.

In Fig 8, queries are on the x-axis and the total message size in MB (Mega Bytes) on the y-axis. On x-axis, for each query we have 2 bars, where first bar (in red) shows the total size of the messages used in our approach and the second bar (in blue) shows the total size of the messages used in general approach.

As we can see in the graph (Fig. 8), the total communicated message size is much less with our approach than that of the state-of-the-art approach. This is because in our approach we do less number of communications while executing queries. Also, we exchange map indices instead of actual join attribute values.

7.1.4 Size of PK-maps and Tuple-index-maps

To show that the size of our PK-maps and Tuple-index-maps grow linearly with the data size, we generated 1GB, 10GB and 30GB data using “dbgen” and then created PKmaps and Tuple-index-maps for each of these data. Size of these maps is shown in Fig. 9.

We can clearly observe from Fig. 9 that, the total size of map structures will be around 10% to 12% of the data size. These maps will then be partitioned and distributed to all the nodes involved in the query execution.
### Query 1: Find the suppliers from ‘EUROPE’, who can supply given part type, and size at a minimum supply cost.

```sql
SELECT s_name, ps_supplycost
FROM region r, nation n, supplier s, part p, partsupp ps
WHERE r_name = 'EUROPE' AND p_type like '%BRASS' AND p_size = 15
    AND r_regionkey = n_regionkey AND n_nationkey = s_nationkey
    AND s_suppkey = ps_suppkey AND p_partkey = ps_partkey
HAVING min (ps_supplycost);
```

### Query 2: Find all the suppliers from nation INDIA, who can supply part named ‘goldenrod’ of size 15.

```sql
SELECT s_name
FROM nation n, supplier s, part p, partsupp ps
WHERE n_name = 'INDIA' AND p_name like '%goldenrod%' AND p_size = 15
    AND n_nationkey = s_nationkey AND s_suppkey = ps_suppkey AND p_partkey = ps_partkey;
```

### Query 3: Find the total number of orders placed by the ‘UNITED STATES’ customers.

```sql
SELECT count (o_orderkey) as TotalOrders
FROM nation n, customer c, orders o
WHERE n_name = 'UNITED STATES' AND n_nationkey = c_nationkey
    AND c_custkey = o_custkey;
```

### Query 4: Find the number of suppliers and customers in AFRICA.

```sql
SELECT count(s_suppkey) as TotalSuppliers, count(c_custkey) as TotalCustomers
FROM nation n, supplier s, customer c
WHERE n_name = 'AFRICA' AND n_nationkey = s_nationkey
    AND n_nationkey = c_nationkey
GROUP BY n_nationkey;
```

### Query 5: Find the suppliers from JAPAN who can supply more than 9900 units of STEEL parts of brand #35.

```sql
SELECT s_name
FROM nation n, supplier s, part p, partsupp ps
WHERE n_name = 'JAPAN' AND p_type like '%STEEL' AND p_brand like '%%#35'
    AND p_partkey = ps_partkey
    AND ps_availqty > 9900 AND n_nationkey = s_nationkey
    AND s_suppkey = ps_suppkey
    AND p_partkey = ps_partkey;
```
7.2 LARGE-SCALE CLOUD NETWORK

We used PlanetLab Cloud for the performance study of this Section. PlanetLab [1] is a group of computers available as a testbed for researchers to develop, deploy and test their applications. It has of many machines in different physical locations throughout the world (called sites). For experiments we used 50 PlanetLab machines located at different parts of the world, running Red Hat 4.1 Operating System. Each machine has 2.33GHz Intel Core 2 Duo processor, 4GB RAM and 10GB disk space. We installed MySQL on all of the machines to perform experiments. All the map structures are loaded into the main memory before each experiment begins.

In this Section, we measure the performance of average query execution time, number of communications made by nodes and the total size of messages exchanged by the nodes. In this experiments we have not used any In-memory database. But, one can take advantage of main memory databases if the data size is huge (in Petabyte or Exabyte) [20] [21] [22]

To perform experiments, we generated 150GB of data using “dbgen” provided by TPC-H benchmark and distributed it to all 50 machines. Each of the 50 nodes contains around 3GB of data. Then, we generated PK-maps and Tuple-index-maps, horizontally partitioned them using the same partition key as to partition data, and distributed these partitions into all 50 machines with corresponding data. We used 5 queries for the performance analysis as shown in Table 4.

Figure 10: Comparison of time taken for query execution between proposed and general approach

7.2.1 Query Execution Time

Comparison of time taken by the queries in Table 4 is shown in Fig. 10, 11. In graphs of Figure 10 and 11, nodes are on the x-axis and time taken (in seconds) on the y-axis. On each graph, red line on bottom shows the result of our approach and blue line on top shows the results of general approach.
Each node has to communicate partial result or the join attribute values with its peers for every join predicate present in the query. Thus each node has to undergo additional message processing. In addition, each node has to wait for the peer message before moving forward to next step in join processing. This will effect the overall time taken by the query. But, with map structures we communicate less number of times and hence, reduce the effect of poor load balancing. Based on the data in Fig. 10 and 11, it is clear that, the query execution time required by the state-of-the-art approach is more than our approach.

7.2.2 Total Number of Inter-node Communications
As explained in the previous Section 7.2.1, each inter-node communication has some inherent overhead. Thus, we also compare the number of messages in addition to the message size.

Comparisons on the total number of messages exchanged during the execution of queries are shown in Fig. 12. In this graph (Fig. 12), queries are on the x-axis and the number of messages is on the y-axis. Based on the result in the graph, it is clear that the number of communications in our approach is fewer than in the general approach. This optimization has a big impact, since the number of communications increases rapidly with increase in number of nodes.

ACKNOWLEDGMENTS

Tingjian Ge is supported in part by the NSF, under the grants IIS-1149417 and IIS-1239176; Benyuan Liu is supported by NSF grant CNS-0953620; and Xinwen Fu is supported by NSF grants 1116644, 0958477, and 0942113.

CONCLUSION

In this paper, we presented two storage structures PK-map and tuple-index-map. In PK-map we store mapping information of primary key and logical record-id of Tuple-index map. In Tuple-index-map, we store mapping of logical to actual record-ids of foreign keys. We then designed an algorithm for processing multi-join queries using the proposed storage structures and reference graph. We analyzed the query processing, storage space required by proposed maps, and advantages of using map in eliminating the data distribution and skew problems. We presented an extensive performance study on small-scale network and large-scale cloud network. For small-scale network, we used 4 virtual node cluster and ran experiments using some of the queries of TPCH benchmark. For large-scale cloud network, we used 50 PlanetLab nodes and ran
experiments to demonstrate the effectiveness of the proposed approach in the real world scenario. Results demonstrate that the proposed approach improves the performance of the ad-hoc multi-join query in Cloud Data Warehouses and reduces communication overhead.

REFERENCE