QUERY OPTIMIZATION FOR CLOUD DATA WAREHOUSE

BY

SWATHI J. KURUNJI
B.E. KVG COLLEGE OF ENGINEERING, INDIA (2006)
M.S. UNIVERSITY OF MASSACHUSETTS LOWELL (2011)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

Signature of Author: _____ **Date:** 11/25/2014

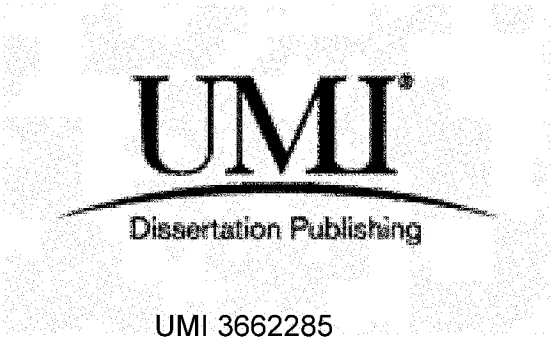
Signature of Dissertation Chair: _____
Name Typed: Cindy X. Chen

Signatures of Other Dissertation Committee Members

Committee Member Signature: _____
Name Typed: Tingjian Ge

Committee Member Signature: _____
Name Typed: William Moloney

Committee Member Signature: _____
Name Typed: Amrith Kumar



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

QUERY OPTIMIZATION FOR CLOUD DATA WAREHOUSE

BY
SWATHI J. KURUNJI

ABSTRACT OF A DISSERTATION SUBMITTED TO THE FACULTY OF THE
DEPARTMENT OF COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

DEFENSE DATE: NOVEMBER 25th, 2014

Dissertation Supervisor: Cindy X. Chen
Associate Professor & Graduate Coordinator, Department of Computer Science
University of Massachusetts Lowell MA

ABSTRACT

Cloud Storage Services have gained a lot of attention from business and academic communities in recent years. They provide dynamically scalable and virtualized resources to consumers, eliminating the hassle of investment and maintenance. Cloud Storage Services are implemented by decoupling different layers of the distributed system, automating the resource scaling, and providing a virtual layer that hides all the underneath system details. A cloud data warehouse is one of the emerging cloud storage services, which has a lot of potential both now and in the future. It has been studied that distributed shared-nothing architecture scales better and is well suited for a cloud data warehouse considering hardware and communication costs.

Analytical applications using a data warehouse tend to have huge table scans, multi-dimensional joins, and aggregates. These join and aggregate operations require communication for transferring data between the nodes that execute a query. In a large cloud data warehouse where the cluster state often changes due to resource availability and Service Level Agreement (SLA), the communication has even greater weight in query costs. Additional metadata maintenance is required whenever there is a change in the cluster. Hence, it requires a robust data model and a distributed query-processing algorithm that works well with such elastic environments.

In this dissertation we study and optimize join and aggregate query processing in a highly distributed shared-nothing Cloud Data Warehouse. Considering some of the

required factors such as shared-nothing architecture, metadata maintenance, and elasticity, we propose a framework with two storage structures and query processing algorithms. The proposed storage structures store relationship information between tables in a star-schema. With the help of these storage structures, proposed query processing algorithms execute a query in such a way that it reduces the interdependency of nodes in the cluster, reduces metadata maintenance, reduces the communication overhead, and improves the overall performance of the query processing. We will then analyze some of the advantages of our framework, such as ability to handle data skew and random data distribution. Finally, we conduct extensive experiments on both small-scale four-node cluster and a large-scale fifty node PlanetLab cluster to validate the effectiveness of our proposed framework.

ACKOWNLEDGEMENTS

I am grateful to Dr. Cindy Chen for being my thesis advisor. She has helped me in every step with her guidance, support and understanding. I have learnt a lot from her, both technical knowledge and professional attitude towards work. When I was studying my Masters, I approached her expressing interest in learning more about database systems and she happily guided me in the right direction. She encouraged me to read research papers and develop analytical thinking in the process. She also gave me the freedom to explore problems and choose my own research topic. I am not sure many graduate students are given such an opportunity to develop their own individuality and self-sufficiency. For everything you have done for me, I thank you from the bottom of my heart.

I would also like to sincerely thank my thesis committee members Dr. Tingjian Ge, Prof. William Moloney and Mr. Amrith Kumar for their valuable discussions and suggestions. Their inputs encouraged me to explore new problems and thus vastly improved the quality of my dissertation work.

I am grateful to the Computer Science Department for giving me the opportunity to study in such a fine environment with great professors, and for supporting my studies through Teaching Assistantship. Thank you Dr. Benyuan Liu, Dr. Xinwen Fu and Dr. Yu Cao for all the support and help.

I would also like to thank all my friends and colleagues who helped me in every

way they can. I cannot forget the time I spent with my friends solving interesting problems and having good discussions.

Finally and most importantly, I would like to thank my husband Vikas Rajegowda and my family for all their love, support and encouragement throughout my studies.

TABLE OF CONTENTS

LIST OF TABLES, FIGURES AND ALGORITHMS

TABLES

FIGURES

ALGORITHMS

1. INTRODUCTION

In recent years, Cloud Storage has become the most attractive and optimal choice for analytical applications that have growing Big-data, and unpredictable need for computing. Cloud Storage Services provide dynamically scalable and virtualized resources to cope with such needs. This is achieved by decoupling different layers of the distributed system such as the physical layer, network layer and logical layer, and then automating the resource scaling process by providing a virtual layer that hides all the underneath system details.

Analytical applications store their data in a data warehouse to achieve good performance for complex read-only queries. Cloud Service providers provide data warehouse solutions along with the storage to help these analytical applications. Such a dynamically scalable data warehouse needs a carefully designed data model and query-processing algorithm. Applying existing distributed data warehouse solutions to a cloud data warehouse will not be efficient. Analytical applications that scale resources often suffer from metadata update and communication overhead problems. This requires re-engineering in the data modeling, indexing, communication and query executing strategies.

In the remainder of this chapter, we first learn two key problems under consideration of this dissertation. We then discuss dissertation goals and an overview of our solutions. Finally, for clarity, we provide background information of related topics.

1.1 PROBLEM ANALYSIS

In this thesis we tackle two of the key problems in data warehouses to make it easily applicable to cloud environments. The first problem is that the meta-data of existing data warehouse databases contain cluster and node specific data table information. Storing such node specific information of the neighboring databases can be costly in a cloud architecture. Due to the cloud's dynamic scale-out and scale-in process, this node specific meta-data information needs frequent updates. This requires more communication in order to maintain meta-data and data consistency. Existing data warehouses reduce some of the query processing cost with the help of data table information stored on their metadata. For example, by storing data type, and key range information on each of the databases, it is possible to redirect data accesses request to specific databases using a unicast communication method. A unicast method will reduce the number of inter-node communications as well as network bandwidth needs. Knowing a neighboring node's data, databases can apply constraints on the request in order to reduce disk access and optimize the data moving cost.

The second problem is that the existing distributed query processing algorithms consider communication costs as a cheaper operation while generating a query execution plan. But, with the virtualization nature of a cloud data warehouse, it is not possible to apply some of the optimizations that the existing distributed systems use. This introduces a greater number of communications, large size of data exchange, and increases overall query processing cost.

1.2 DISSERTATION GOALS AND SOLUTION OVERVIEW

It is not possible to solve the above mentioned problems by optimizing a query processing algorithm alone. It also needs revising the metadata model and communication method. So, in this thesis we optimize both the metadata model and communication overhead along with the query-processing algorithm to decrease the overhead caused by the cloud architecture.

The first goal of this thesis is to decrease the metadata maintenance. To do this, we only maintain cluster change information in the metadata. We do not maintain data distribution information of each node in the cluster. With this, we eliminate the data distribution metadata maintenance and its associated communication. Since we do not have data distribution information, it is not efficient to use a unicast method for communication. Instead, we will use a broadcast method to do inter-node communication.

By literature review on cloud infrastructure products, we got to know that the broadcasting method is more efficient than sending messages to specific nodes in the network. Each node does not have to know where the remote data is stored or which table partition is stored on which node of the cluster. Instead, each node can broadcast the intermediate results of a query to all other nodes in the cluster. This eases the scale-out and scale-in of the cluster. To decrease the communication overhead caused by the broadcasting method, we will use special storage structures (PK-map and Tuple-index-map), which are explained in Chapter 3.2.

The second goal of this thesis is to reduce the interdependency between the nodes

executing the query. Reducing interdependency reduces the number of inter-node communications and reduces the size of inter-node messages exchanged during the query execution. Interdependency between nodes is mainly caused due to data table sharding in the distributed relational database system. Some query operations such as the distributed-join and distributed-aggregate operation requires all partitions of data tables stored in neighboring nodes to finish their execution. Hence, nodes need to exchange data in order to complete these operations.

We reduce this interdependency and communication overhead with the help of our proposed storage structures (Chapter 3.2). These structures allow queries to perform join and aggregate operations independently on each local node. In addition, nodes only exchange record ids of the result when the query needs other attribute values of the remote data table. Hence, only relevant data (record ids and some header information) is exchanged on each communication. This reduces the size of the message exchanged.

Our proposed map structures, PK-map and Tuple-index-map, stores relationship information of the data tables in the star-schema. These map structures are sorted on primary and foreign key attribute values. This ordering allows us to apply run-length encoding and reduce the size of the map structures.

These map structures are created for every primary key that is present in the local node of the cluster. It contains information required for the local node to perform join and aggregate operations independently and efficiently. With the help of this information most of the join and aggregate operations can be done in parallel on each of the nodes without any communication.

We then designed a query-processing algorithm, which uses proposed storage

structures and referential integrity constraints to generate an optimal plan to process the query. This algorithm executes a query as independently as possible with minimal communication overhead. Details are shown in Chapter 3.3 and Chapter 3.4.

In addition, we improve the aggregate query performance by eliminating most of the group-by operations required for aggregate operation. Because map structures are sorted on primary and foreign key attributes, most of the group-by required by aggregate operation of the query is performed in the early stage along with map scan. We maintain the sort order of partial results while processing the query operations in order to reduce the aggregate operation work and time.

For performance analysis, we first used a small-scale private virtual network from the Computer Science Department at UMass Lowell. We ran some multi-join TPC-H benchmark queries to prove our concept. We then conducted extensive experiments using a large-scale PlanetLab network. Experiments show how our proposed framework improves the performance of join and aggregate queries, decreases node-to-node communication overhead, and reduces database table access required for the query in Cloud Data Warehouses.

1.3 BACKGROUND CONCEPTS

Data Scientists and Analysts use analytical applications to discover useful pattern in the data and perform report generation. Then they use these reports to suggest solutions, which will help in decision-making at various levels of the business.

Analytical applications are designed using various techniques and architectures to provide fast data access and quick report generation. They use ETL (Extract Transform

and Load) techniques to pull data from various data sources, extract useful information from those data, and store them in an organized manner to support fast data access. Analytical applications run read-only queries to generate reports and they use data warehouses as a storage architecture for this purpose.

Data warehouses are designed to accommodate Online Analytical Processing (OLAP) tasks. They are read-optimized and store historical data. The interval between historical and current is purely dependent on applications. For example, some applications use a write-optimized database to store all the insert/update/delete operations and periodically move those data to data warehouses. Historical data may come from a variety of fields, such as the business history of a supermarket, historical data of credit card transactions, biological data of diseases, scientific data, or agricultural data. Queries run by the analysts on these data warehouses are iterative, complex, read intensive, and compute intensive [5].

There are several good commercial OLAP (Online Application Processing) applications for data warehouse analysis such as EMC Greenplum [3], HP Vertica [10], Actian Paraccel [49] [50], SAP HANA [51], IBM InfoSphere [4], Microstrategy [6], OracleBI [7], and SqlServer [8]. But these were primarily designed for general distributed database systems and not for cloud architecture. A survey conducted by Oracle [39] shows that 36% of Data Warehouse users are having performance problems. Common performance bottlenecks include loading large data volumes into a data warehouse, poor metadata scalability, running reports that involve complex table joins and aggregation, an increase in the complexity of data (dimensions), and presenting time-sensitive data to business managers etc.

Data in the data warehouse is logically organized using a star-schema data model as shown in TPC-H [9] and DSS [71] benchmarks. Star-schema consists of one or more fact tables referencing multiple dimension tables. The star schema is a special case of the snowflake schema and is very effective for handling multidimensional analytical queries. Dimension tables are usually smaller in size, which stores the descriptive data such as the movie, actor, director, or genres table in Figure 1. On the other hand, fact tables are very large and they store data of the business transactions such as the sales table in Figure 1.

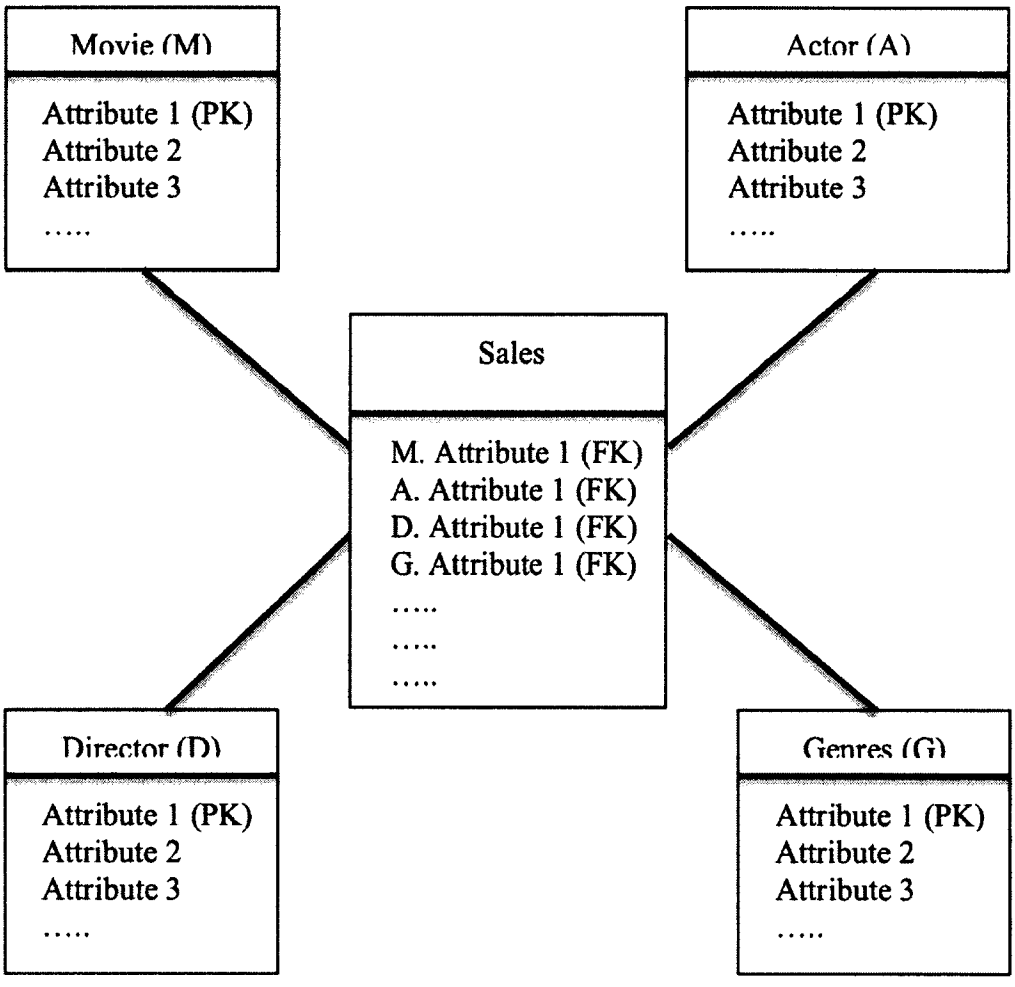


Figure 1: Star-schema of a movie database

Figure 1 is a movie database, which stores movie and its sales information for further analysis. Here, *Sales* table represents a fact table of the star-schema. Sales table is often updated and grows in size for each movie sale. On the other hand, movie, actor, director, and genres tables represent dimension tables of the star-schema.

Physical organization of data in data warehouses takes many forms. Organization of data may be row-oriented, column-oriented or hybrid, depending on the type of the application, usage or queries. Row-oriented databases store rows of a table sequentially on physical storage (disk tape) as shown in Figure 2. These databases can be updated easily and quickly. The disadvantage of a row-oriented database is that the read performance is poor when we have a flat table. This is because the database will read a whole row even though the query requires only a few attributes of the table.

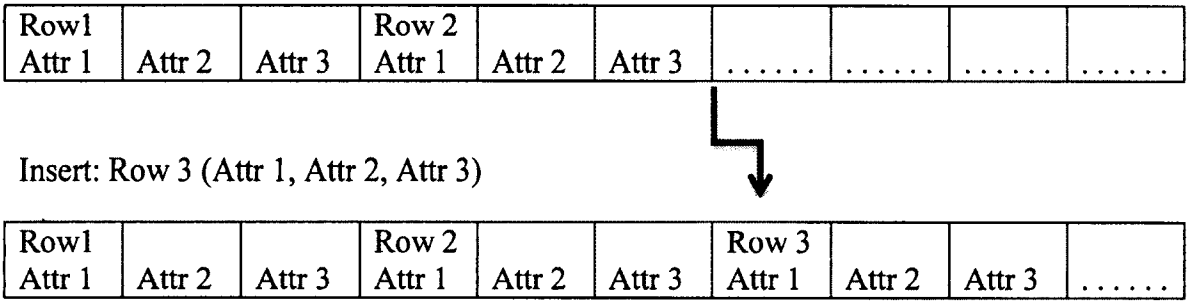


Figure 2: Row-Oriented Database Disk Tape

Column-oriented databases store attribute values of one or more columns (called projection) sequentially on physical storage as shown in Figure 3 [19] [21]. These databases can access data quickly from a flat table. When the query needs few columns of the table, the database will only read those attributes and not the whole table. This is possible because the attribute values of different rows are stored sequentially (Figure 3). The disadvantage of a column-oriented database is that the write performance is poor.

This is because, for each insert operation, the database will write in many places as shown in Figure 3. Another disadvantage is that the performance starts degrading, as a query requires a greater number of attributes of a single table. Since we store each column data separately on disk tapes, we need to perform join between columns of the same table to get the complete table data.

Different analytical tools use different types of physical and logical organization of data in order to achieve different performance advantages.

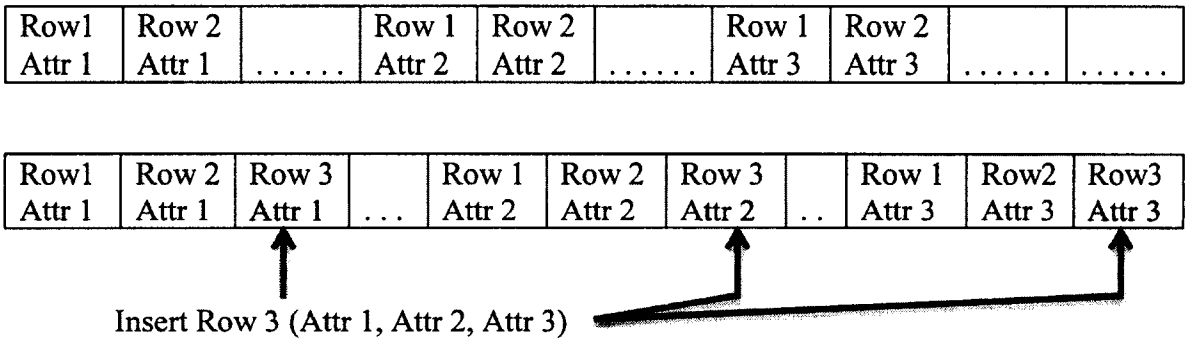


Figure 3: Column-Oriented Database Disk Tape

Data Warehouses or decision support systems use join, group-by, and aggregate operations very often in formulating analytical queries. These queries are used to perform different real world operations such as prediction processing, analytical processing or data mining. For example, using our movie database of Figure 1, if we want to answer query Q1, we need to access genre, sales and actor tables.

Q1: "Find the actor who generated maximum revenue for his/her movie in some specific genre"

Each node of the distributed database cluster needs to perform the following steps to answer the query Q1:

a. Find all IDs from the genres table for a given genre type "XXX"

b. Communicate the results with the neighbor database nodes in the cluster

c. Join the resultant genre IDs with the sales table to get the rows matching genre "XXX"

d. Find the local maximum revenue sales row

e. Communicate the results with the neighbor database nodes in the cluster

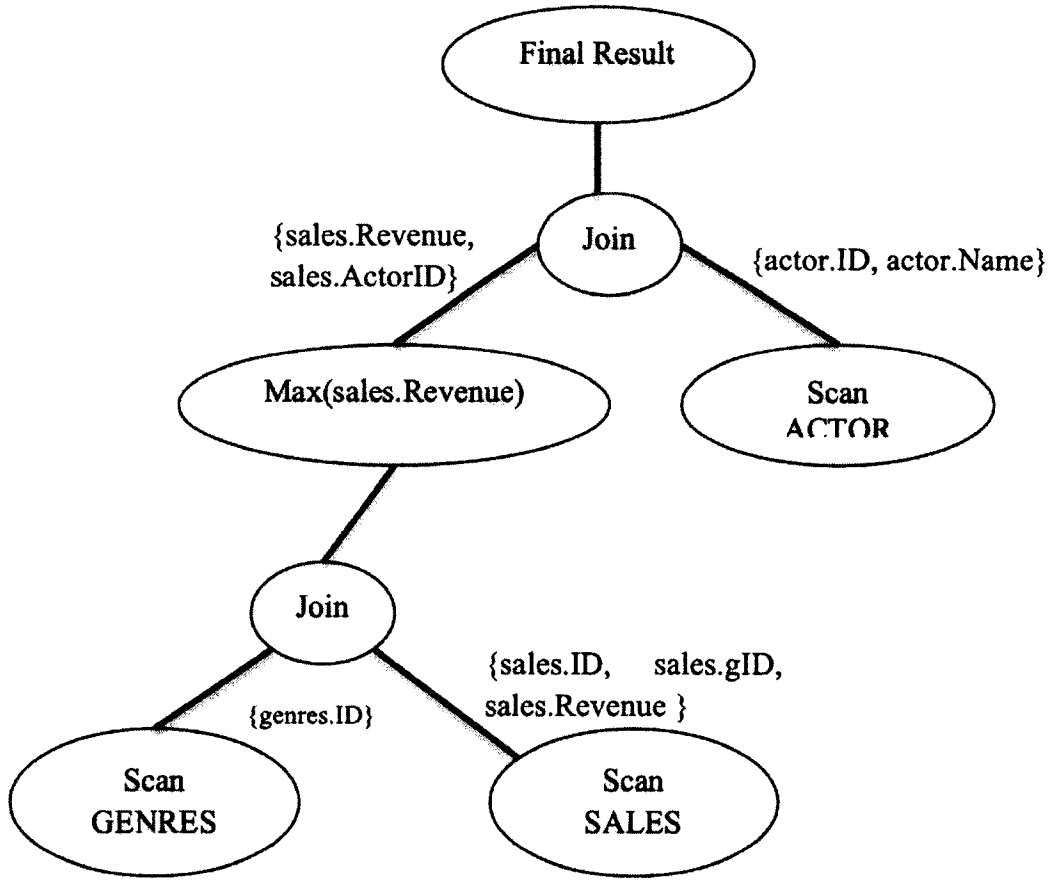


Figure 4: Query plan for query Q1

f. Join the resultant sales ID with the actor table to find the local final result

g. Communicate the final result

Efficient execution of such analytical queries in large-scale and dynamic Cloud databases is a challenging problem [31]. One of the main reasons is that we need to update global and local indexes (such as Distributed Hash Tables) every time the data is moved (or changed), or a new machine is added. For example, in a ring network like the Cassandra architecture, new machines are added to the ring near the bottlenecked machine, and the data is redistributed between that machine and its new neighbor. After the redistribution, metadata also needs to be updated so that future queries are directed to the right node. In a cloud architecture, such redistribution and updates happens very frequently. Another reason is that the nodes have to exchange intermediate result data for every distributed join or aggregate operation as we saw in the Q1 execution steps.

In traditional distributed databases, to reduce the exchange of intermediate result data during the execution of a query, tables are horizontally partitioned on the join attributes and related partitions are stored on the same physical system. This architecture causes data skew and load imbalance between the nodes in the cluster (Chapter 5). There has always been a tradeoff between data distribution and message exchange during query processing. In a cloud environment, it is not possible to ensure that these related partitions are always stored on the same physical system due to the virtualization, Service Level Agreement (SLA) [55] and resources availability. Before we analyze query processing in cloud architectures, let us learn the basics of cloud computing, cloud storage, and some of its services.

Cloud computing and storage has gained attention from researchers and

consumers in recent years. It is designed to provide dynamically scalable and virtualized resources to consumers, eliminating the hassle of investment and maintenance. This is achieved by decoupling different layers of the distributed system such as the physical layer, network layer and logical layer. Automating the resource scaling and providing a virtual layer that hides all the underneath system details. Commercial cloud products also provide each layer of system services separately so that customers can mix and match their requirement.

Many commercial products from Google, Amazon, Microsoft, EMC, IBM, etc., provide cost effective solutions like hourly, monthly or yearly usage billing [1] [2] [3] [4] [5]. Some of the services provided include, Compute Services, Storage Services, Database Services, Analytics Services, Application Services, etc. Customers can choose whichever option best fits their requirements [54].

Compute Services provide dynamically scalable compute capacity in the cloud. This provides virtual machine images of various platforms to users over a user-friendly web console. Some of its features are on-demand scalability of virtual machine images, security with various types of access control configurations, cost effective billing options, and flexibility of architecture (zones/regions). Some of the commercial products are Amazon Elastic Compute Cloud (EC2), Google Compute Engine (GCE), and Windows Azure Virtual Machines.

Storage Services allow users to store and retrieve any amount of data, at any time, from anywhere on the web. They provide the storage scalability, various replication strategies, different access policies, and degrees of data encryption and data consistency. Some of the commercial products are Amazon Simple Storage Service (S3), Google

Cloud Storage (GCS), and Windows Azure Storage.

Database Services allow users to set-up and operate relational and/or non-relational databases in the cloud. This service reduces the Database Administration tasks (DBA). Some of the commercial products are Amazon Relational Data store (RDS), Amazon DynamoDB (NoSQL), Amazon Redshift (Column-store Data warehouse), Amazon ElasticCache (In-memory cache), Google Cloud SQL (MySQL version), Google Cloud Datastore (Key-Value store) [64], Windows Azure SQL Database, and Windows Azure Table Service (NoSQL).

Analytics Services allow users to analyze massive amount of data sets stored in the cloud either in cloud storage or cloud databases, using programming models. Applications can perform data-mining, log file analysis, machine learning tasks, web indexing, etc. Some of the commercial products are Amazon Elastic MapReduce, Google MapReduce Service, Google BigQuery, and Windows Azure HDInsight.

Application Services provide application runtimes and frameworks service, queuing service, email service, notification service and media service. Some of the products of Amazon are Amazon Simple Queue Service (SQS), Simple Email Service (SES), Simple Notifications Service (SNS), and Elastic Transcoder. Products of Google include Google App Engine, Task Queue Service, Email Service, Cloud Messaging, and Images Manipulation Service. Windows products include Windows Azure Web Sites, Queue Service, Notification Hubs, and Media Services.

The 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 networks. Applying existing distributed query processing, which is designed with the consideration that the cluster status does not change often, is not optimal for the cloud architecture. In addition, existing distributed DBMS use a two-phase query plan generation algorithm. This algorithm generates query plans optimizing the disk access in the first-phase. Then, they optimize those plans to reduce communication cost. But, communication cost is also an important factor and it requires more weightage due to the cloud's elasticity and virtualization architecture.

In the cloud architecture, data is distributed to different nodes depending on the nature of an application, and the availability of resources such as storage space and CPU cycles. In addition, the physical location of the data may dynamically change from one node to another to meet customer SLA (Service Level Agreement) [55]. This increases node-to-node communication overhead due to metadata maintenance, node tracking in the cluster and data distribution on each of the nodes.

For example, we have 4 nodes (n1, n2, n3 and n4) in the cluster (Figure 5) and a network monitoring system (NMS) to maintain the SLA [55]. Suppose, n2 begins to slow down and our network monitoring system identifies that it might not meet the SLA of some customer c1. It will initiate the data movement of c1 from n2 to some new nodes, n5 and n6 that contain the required space and other resources. Now, we need to update the cluster metadata by deleting n2 and inserting n6 and n7. The resulting network after scale-out is shown in the middle component of Figure 5. Also, we need to update the data distribution on each of the new nodes added to the cluster so that future messages are re-directed properly (Analysis is in Chapter 5). This will increase the network bandwidth as well as the CPU usage cost of the customers. In addition, each node has to communicate

with four other nodes instead of three for distributed join or aggregate processing.

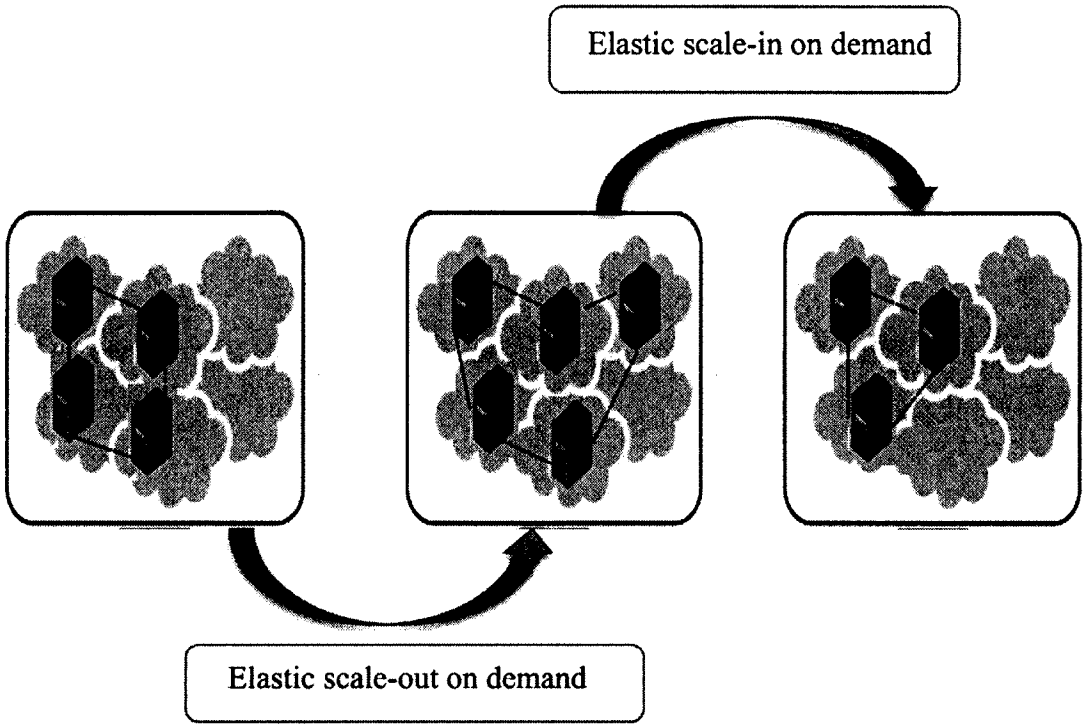


Figure 5: Cloud Network

As the resources are scaled-out, the number of nodes involved in the execution of a query increases and this results in increased node-to-node communication. This becomes critical when we have huge data (in Terabytes or Petabytes) stored across a large number of nodes. Execution of analytical queries in such a cloud data warehouse becomes more complicated when queries contain multiple joins between partitions of different tables stored in different nodes. These joins need back and forth communication of data 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. When the number of joins in a query increases, the

performance will further degrade. Each increase in communication will increase network bandwidth usage and CPU usage for message processing.

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 and reduce the size of the partial query results exchanged in each communication. In this dissertation we solve the above problems by providing storage structures and a query execution strategy, which allows nodes to execute queries as independently as possible. We also exchange the minimal amount of data required by the database nodes during the query execution. We will see the details in coming chapters.

2. LITERATURE REVIEW

The aim of this Chapter is to review some of the research work that proposes key features required for cloud data warehouses. We then, review state-of-the-art distributed join processing algorithms and the optimizations proposed by researchers. Finally, we will review state-of-the-art aggregate query processing algorithms and their existing optimizations.

2.1 CLOUD DATA WAREHOUSES

Cloud Computing Services emerged in the year 2006 when Amazon implemented its commercial cloud product, Amazon Web Services (AWS). From then onwards, there has been a great deal of discussion in the industry and academia about cloud computing and its services. National Institute of Standards and Technology (NIST) published 16 versions of special publication between 2009 and 2011 [60], which characterize important aspects of cloud computing and provide a baseline for discussion from what is cloud computing to how to best use cloud computing. The NIST definition of cloud computing is:

"Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider

interaction"

Cloud computing and associated Analytics has been one of the top 10 strategic technology trends of Gartner since 2008 [61][62]. Gartner suggests some of the improvements that need to be incorporated for his listed technologies. In the top ten list of trends for 2015 he speaks about cloud computing as,

"Network and bandwidth costs may continue to favor apps that use the intelligence and storage of the client device effectively, coordination and management will be based in the cloud"

This means that the network and bandwidth cost on the server side is not robust enough. Optimizing network strategies alone cannot solve this problem. We will also have to join hands with the surrounding areas which use this network such as storage, databases etc. The requirement of optimizing network and bandwidth cost is also stated in Berkley RAD Lab report [63]. Table 5 of [63] compares the costs (per month) of bandwidth usage, CPU, and disk storage between years 2003 to 2008. It shows how CPU and disk storage cost has decreased while bandwidth usage cost has increased tremendously.

Over the years many companies became big players of cloud computing services such as Amazon [1] [65], Google [2] [64], Microsoft [5], EMC [3], IBM [4], Yahoo [73], and HP Vertica [10]. We will see some cloud database products and their key features.

Amazon DynamoDB is a schema-less NoSQL database service that is highly scalable and fast. It supports both document and key/value storage models. It is fully distributed with shared-nothing architecture providing functionalities like unlimited storage, automatic scaling, and automatic replication (for fault tolerance, availability).

DynamoDB uses a consistent hashing technique to distribute the records to the cluster nodes. Each node is assigned a position (hash partition range) in the cluster [56]. During insertion of records, a hash function is applied on the primary key. Then, the record is sent to the node that is responsible for the hash range where the hash result value falls.

Amazon Redshift is a column-oriented cloud data warehouse solution that can scale to petabytes. Redshift is composed of amazon's cloud infrastructure components and ParAccel's database system in the backend. Redshift has a collection of computing resources called *nodes*, which are organized into a group called a *cluster*. Each cluster runs an Amazon Redshift engine and contains one or more databases. Databases are relational with a shared-nothing architecture. It can store data in a compressed format.

Google provides two kinds of database storage services called *Google App Engine Datastore* and *Google Cloud SQL* [64]. *Google App Engine Datastore* is a schema-less object datastore. It internally uses Bigtable, which in turn uses the Google File System (GFS) on the physical level to store the data. GFS is a NoSQL key-value store with master-slave architecture. With this master-slave architecture, it provides the scalability feature of cloud service. The slave servers (called tablet servers) can be dynamically increased or decreased depending on the workload. The master is responsible for handling scalability and other slave server management functionality. Fault tolerance is achieved by replicating the data into multiple data storage servers. Google provides a SQL like interface to the users to run their query called Google Query Language (GQL). It supports strong consistency for read queries and eventual consistency for other queries like insert/delete/update operations. For read queries it retrieves data from the primary storage with the most recently updated data. But, for other type of queries it will retrieve

data from secondary storage location if the primary storage is unavailable.

Google Cloud SQL is a MYSQL database service provided over the web. It has all the capabilities and functionalities of MYSQL with additional features like easy to use MYSQL client, synchronous and asynchronous replication across multiple zones for greater availability and durability, customer data encryption for more security, etc.

There are many other private and open source companies who have competitive cloud products such as Cloudera, Snowflake Computing, Cassandra, HBase, Hive, CouchDB, HadoopDB, and HyperTable.

Around 2009 and later many researchers have analyzed database architectures in detail and proposed changes to meet the goals of cloud services such as elasticity, virtualization, and security [11] [14] [68] [69]. Abadi [11] explores the limitations and opportunities of deploying data management systems on the cloud computing platforms. According to the author, some of the key features that are required for the cloud services to be able to work efficiently are elasticity, security and replication. Implementing a parallelizable shared-nothing architecture can make a database cluster easily scalable and provide replication. We will skip the security features here since it is not the concern of this dissertation. So, the author claims that the data analysis tasks, decision support systems, and application specific data marts are best suited and can easily be ported to cloud computing and storage services. On the other hand, transactional database systems are not yet suited for cloud architecture because of the difficulty in supporting the ACID property and the difficulty in providing a shared-nothing architecture. Finally, they point out some of the open problems in the existing cloud architecture such as hybrid architecture (with Map-reduce and shared-nothing features), and check pointing of

intermediate results in order to provide fault tolerance.

HadoopDB [14] proposes a hybrid architecture combining the positives of both parallel and MapReduce systems. This is designed for cloud-based data analytics incorporating the requirements of cloud architecture. Parallel databases provide the high performance and scalability needed for large-scale data analysis. But, it doesn't handle fault tolerance and lacks the ability to operate in a heterogeneous environment. On the other hand, MapReduce systems work well in heterogeneous environment and handle fault tolerance very well. But, MapReduce uses a one-time query processing model where we cannot take advantage of data modeling and pre-loading of data during query execution. So HadoopDB uses the Hadoop's scheduler, job tracker and communication layer to have fault tolerance and heterogeneity. Then it uses multiple single database nodes and connects them to Hadoop framework to get performance benefits of parallel databases. HadoopDB executes most of the single node query processing work inside the databases. Results of this paper show that Join and Aggregate operations perform better in HadoopDB than Hadoop because HadoopDB takes advantage of indexing provided by the database layer. But, HadoopDB is still not as powerful as distributed shared-nothing databases like HP Vertica and others. In the next section, we will look at some of the join query processing algorithms and their optimizations.

2.2 JOIN QUERY PROCESSING

The join operation was introduced by Edgar Codd when he coined the term "relational database" in his research paper [66] in 1970. The main goal of designing a relational database model is to detach the dependency of user applications from the data

storage. According to him, by removing this dependency we could allow user applications and/or data storage to change in parallel without harming one another. Edgar also wanted to make data storage well organized with less redundancy to improve the storage and query performance. So, he introduced a relational database model containing relations and a unique-id representing each record in it. These related relations have common keys through which users can join two relations and get a ternary relation.

Later researchers started proposing different join processing algorithms based on the type of join (inner / outer / self), data size, and processor memory. State-of-the-art join processing algorithms are Nested-Loop-Join, Index-Join, Sort-Merge-Join and Hash-Join. These algorithms have their own advantages and disadvantages. The query planner of the database engine chooses the join algorithm based on the available indexes, availability of memory, parallelizability, etc.

Many researchers proposed variations for the above state-of-the-art join processing algorithms in order to reduce the time taken for query processing. They proposed algorithms using parallel and distributed techniques. Pipelining the join input relations is one of the commonly used techniques to reduce the distributed query processing time. We will discuss some of them below.

Another popular join algorithm called Semi-Join is proposed to improve the bandwidth usage by reducing the amount of data transferred between nodes executing the query in a distributed environment. In this algorithm, nodes executing the query first exchange only join attribute values of one of the input relations. Then, each node performs a local processing using exchanged values and a second input relation to get the matching records. After local processing, nodes exchange these matching records of the

second relation with neighboring nodes. Finally, nodes perform a join operation of the first relation and the exchanged second relation records. As we can observe here, the Semi-Join algorithm reduces the number of rows of the second relation before data exchange. This reduction is advantageous when the second input relation size is larger than the join attribute values of the first relation. Looking into the rate of increase in the complexity and size of real world data, it is attractive to choose this algorithm to reduce the overwhelming communication costs. But, the problem is that this algorithm performs two join operations instead of one in order to reduce the number of rows exchanged. Thus, it is a tradeoff between local computation and bandwidth usage.

The Skalla system [15] is designed for efficient evaluation of OLAP queries in a distributed data warehouse. It is specifically designed to perform analysis of flow level traffic in the IP network. The Skalla system has a central master coordinating the query execution, and many slave nodes attached to network routers. These slave nodes are responsible for distributed query processing. The master node handles the local to global data mapping. The Skalla uses indexing and coalescing techniques to improve the query performance. Experiments with the Skalla system show the advantages of using materialization techniques to improve the query performance. This system cannot be directly incorporated to a cloud environment because of its meta-data maintenance problem and single point failure. But, network traffic analysis is one of the areas where we can use cloud-computing services to improve the network utilization.

Efficient ways to concurrently execute multiple queries in order to increase the throughput is discussed by Papadias [20], Raman [67], and Candea [16]. Papadias [20] optimizes the existing multi-query processing algorithm that generates one global query

plan by adding queries need to be executed one by one from the queue. The proposed approach uses a Best View First (BVF) technique to choose the optimal query plan for execution that covers maximum number of queries in the queue with optimal cost. At each stage, it finds the cost for subset of queries using existing materialized views and then adds the best one to the global query plan for execution.

On the other hand, Raman [67] argues that the query processors should aim for constant response time for all queries, with no assumption about tuning. Because the database query performance is purely dependent on available indexes, materialized views etc., some queries take more time than others making it less attractive for analytical users. So, he proposes a model called BLINK, which gives consistent response time to all the ad-hoc queries. The Blink runs every query as table scans over a fully de-normalized database, with hash group-by done along the way. To perform scan efficiently, it uses partitioning and compression schemes. It runs range and equality predicates over compressed data and uses an efficient hash-based aggregation within the L2 cache. De-normalization will only work for small databases and will work for an application that has few attributes or features. The problem with the de-normalization is that it does not work well for large databases because of its increase in data redundancy and consistency. It also won't work for analytical queries that need few columns.

Candea [16] has a different view and he proposes the C_{JOIN} model in order to execute multiple queries simultaneously by generating a single query plan and by sharing I/O, computation and tuple storage required for all the queries. It uses unique-id for differentiating different query and its predicates. For example c_{ij} is a selection predicate of a Query Q_i on the dimension table D_j. The continuous stream of scanned fact table is

given as input to the C_{JOIN} architecture. This fact table is passed through a sequence of filters before sending it to an aggregate operator for output generation. These filters correspond to dimension table predicates of all the queries. In this way it shares the I/O and computation among all the queries. Experiments show that it executes more number of queries at a time compared to the existing databases. In addition, the query response time is not increased as it in the existing databases (query time linearly increases as more queries are added).

As the size of data used by analytical databases started to increase in big numbers, some of the researchers at MIT were thinking to change the physical architecture of the databases to improve the performance of analytical queries. They analyzed the type of query, amount of data they retrieve from data table, etc. Finally, they designed a read-optimized distributed database management system, C-store [21], which stores the data in a vertical storage structure in the physical layer. There were commercial column-oriented databases such as Sybase-IQ [19], which was one of the first column-oriented DBMS. But in late 2005 people started to see the importance of column-oriented databases and column-oriented databases gained a lot of attention in data warehouses. This vertical storage structure has improved the performance of data warehouse applications in the order of magnitude by reading only relevant data from the disk and incorporating parallelization techniques.

Several papers were published studying these column-oriented databases. One of them is [12], which analyzes the internals of the column-oriented DBMS and states some of the open research problems in the context of column-store systems, including the physical database design, indexing techniques, parallel query execution, replication, and

load balancing. Another paper Column-Vs-Row [13] compares the actual difference between the column and a row-oriented DBMS in both physical and logical levels.

Holloway and DeWitt [17] compare the advantages and disadvantages of the column-oriented and row-oriented database systems by analyzing some of the features like join operation processing. The join processing is compared based on the % of tuples retrieved for the query, and the number of columns of table present in the query. It is noticed that, when the number of columns required by the query is small, column oriented databases perform very well. But, as the number of columns required reaches to four, they see the performance deterioration due to stitching of columns to form a table. This paper proposes two techniques for column-oriented databases to solve the problem of reorganization overhead during the stitching of columns to tuples or other query operations.

Ivanova [18] improved the performance of column-oriented databases by proposing *adaptive segmentation* and *adaptive replication* strategies. An adaptive segmentation splits the columns into non-overlapping segments based on the query load. Similarly, adaptive replication creates replicas of these segments and store in different order to reduce the reorganization cost. These proposed techniques trade storage space for the query performance.

Increase in size and complexity of the data in modern analytical systems, and the emergence of cost efficient cloud computing services has attracted MapReduce based system researchers. They started to analyze the requirement of analytical applications and started to incorporate those requirements using MapReduce framework and cloud computing services such as scalability, fault tolerance, etc. Some researchers argue that

data analysis workloads tend to contain huge table scans, multi-dimensional joins, and aggregates, which are easy to parallelize across a shared-nothing architecture. Paper [68] states that the shared-nothing architecture scales best when we consider hardware cost into account (Teradata, Oracle's Exadata). But, the existing parallel databases were not optimal for scaling because it assumes that failure is a rear event, and it needs homogeneous databases. Some other researchers argue that MapReduce based systems are best suited for data analysis because it can scale to thousands of nodes, it is cheap, and it handles fault very well. The problem with MapReduce based systems is that it is difficult to execute queries of structural database systems [69]. Because MapReduce based systems are not designed to handle structural data analysis and they won't scale well in such cases.

HadoopDB [14] proposes a hybrid architecture combining the positives of both parallel and MapReduce systems. Parallel databases provide the high performance and scalability needed for large-scale data analysis. But, it doesn't handle fault tolerance and lacks the ability to operate in a heterogeneous environment. On the other hand, MapReduce systems work well in a heterogeneous environment and handle fault tolerance very well. But, MapReduce uses a one-time query processing model where we cannot take advantage of data modeling and pre-loading of data during query execution. So HadoopDB uses the Hadoop's scheduler, job tracker and communication layer to have fault tolerance and heterogeneity. Then it uses multiple single database nodes and connects them to Hadoop framework to get performance benefits of parallel databases. HadoopDB executes most of the single node query processing work inside the databases. Results of this paper show that Join and Aggregate operations perform better in

HadoopDB than Hadoop because it takes advantage of indexing provided by the database layer. But, HadoopDB is still not as powerful as distributed shared-nothing databases like HP Vertica and others. In the next section we will look at some of the aggregate query processing algorithms and their optimizations.

2.3 AGGREGATE QUERY PROCESSING

Aggregate query processing has been studied in many research works [29]. But, as per our knowledge, not many of them consider communication cost in optimizing the aggregate query processing. Some of the earlier papers, which optimize the aggregate query processing, are [26] [38] and [45]. These papers provide optimizations by pushing down group-by in the query tree to improve the query response time.

Eager-Lazy-Aggregation [45] proposed two kinds of transformations namely, eager aggregation and lazy aggregation. In the eager aggregation, group-by operation is pushed down in the query tree such that the group-by is performed once before the join operation and once after the join. This aggregation reduces the size of the input to the second group-by operation resulting in a better overall plan. Lazy aggregation is the opposite of the eager aggregation and it can be useful in the presence of views or pre-computed aggregate results. On the other hand, Include-Group-by [38] paper presented a greedy conservative approach and an extended annotated join tree method to push down the group-by operation and optimize the query processing. We use the above transformations of [45] in our system along with our PK-map and Tuple-index-map to generate optimized query plan to process aggregate queries.

Order-Optimization [28], try to push down sorting in the query tree and present

techniques to reduce the number of sorts needed for query processing by finding the cover set using keys, predicates and indexes. Since our proposed map structures are already sorted on keys, and we scan our maps rather than tables to perform join operations. We eliminate most of the sort operations required for join operation on the tables.

Coloring-Away [46], proposes the query plan generation using a tree-coloring mechanism. During the query processing, both communication cost (which includes time to transfer and process transferred data) and data re-partitioning are inter-dependent, and contribute towards the overall query cost. Hence, the paper considers both of these measures and uses tree coloring to generate an optimal query plan. In our framework, we optimize the query operations that cause the above mentioned query performance problems such as aggregates and joins by doing the sort and group-by on the fly.

Avoid-Sort-Groupby [47], proposed a query-plan refining algorithm through which unnecessary sorting and grouping can be eliminated from the query plan. It uses inference strategies and order properties of the relation table to find the unnecessary sorting or grouping. The authors present a mechanism to combine both ordering and grouping using finite state machines. T. Neumann [43], points out that it is necessary to consider both ordering and grouping to generate the query plan. He explains the difference between ordering and grouping; ordering is the sequence of attributes whereas grouping is the set of attributes. The authors present a mechanism to combine both ordering and grouping using finite state machines.

Cooperative-Sort [48] [52], presented an evaluation technique for sorting tables. This technique is for those queries that need multiple sort orders of the same table on

different attributes. This technique creates partitions or chunks by sorting on the first attribute. Very small sequential partitions are then joined to form composite chunks of up to size M (size of memory). Then, these chunks are sorted on the second attribute. This minimizes the I/O operations of the successive sort operations, which reduce the overall query cost.

Pre-computing the aggregates is proposed by many other researchers [30] [40], which are useful for decision support systems. Decision support systems store huge amount of historical data for analysis and decision-making. These databases are updated less frequently (once a hour/day) on batches. This made it easy to compute the aggregation ahead of time and store it as data cubes or materialized views. Recently, the interval between historic and current data has been reduced a lot. This will make it complicated and time consuming to re-compute the data cubes or materialized views every time data gets updated. Recent research by companies like HP, Oracle and Teradata [32] [42] [44] shows new parallelization schemes for processing join and aggregate operations, eliminating the data cubes.

In this dissertation, we concentrate on the join and aggregate query executions in a cloud environment without any pre-computation of queries (data cube or materialized views). Through the proposed framework we ensure random data distribution and thus solve data skew problem. Techniques used in this dissertation decrease the metadata maintenance during cluster changes (scale-out / scale-in). We will see in detail in coming chapters.

3. PROPOSED FRAMEWORK

With the understanding of the problem statement and its background, we now focus on our proposed framework. In this chapter, first we will look at the star-schema benchmark that we are going to use in the rest of this dissertation for analysis and experiments. We then explain the proposed storage structures and its application using TPC-H Star-schema. With the understanding of storage structures we will look at the Join query processing, its algorithm and a case study. Finally we will look at aggregate query processing and its algorithms in detail.

3.1 TPC-H BENCHMARK SCHEMA USED FOR ANALYSIS

Star and Snowflake Schema representations are commonly used in the read-optimized Data Warehouses. The star-schema is a special case of the snowflake-schema and is very effective for handling multidimensional analytical queries. The star-schema consists of one or more fact tables referencing multiple dimension tables. Dimension tables are usually smaller in size, which stores the descriptive data. In Figure 6, dimension tables are REGION, NATION, SUPPLIER, CUSTOMER, PART, and PARTSUPPLIER. On the other hand, fact tables are very large and they store data of the sales transactions. ORDERS and LINEITEM tables of Figure 6 can be considered as fact tables of the TPC-H benchmark schema.

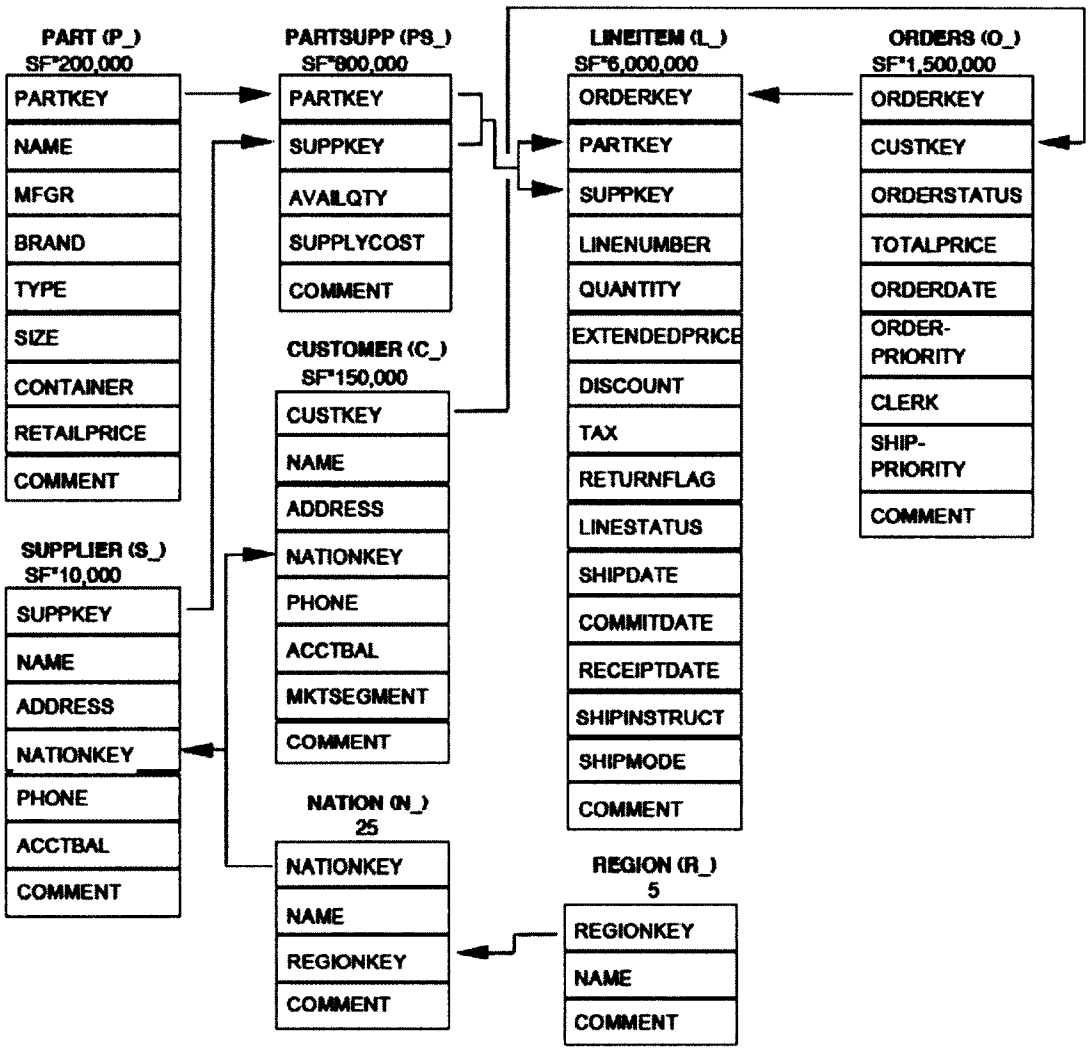


Figure 6: TPC-H Star Schema from TPC BENCHMARK H Standard Specification Revision 2.15.0 (Source: [9], page 13)

In this thesis we use the star-schema (Figure 6) from "TPC BENCHMARK H Standard Specification Revision 2.15.0" for analysis and performance evaluation. The 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. The TPC produces benchmarks that measure transaction processing (TP) and the database (DB) performance in terms of how many

transactions a given system and a database can perform per unit of time. These benchmarks are classified based on the type of database applications like, TPC-C and TPC-E is designed to evaluate OLTP (Online transaction processing) applications. TPC-H is a decision support benchmark, which is designed to evaluate the functionalities of business analysis applications (Online analytical processing applications). It provides a star-schema and a set of business queries particularly designed to exercise the functionalities of the system under test in a realistic context, portraying the activities of a wholesale supplier.

Figure 6 is a schema of an industry, which must manage, distribute and sell its products worldwide. The number/formula below each table name represents the cardinality (number of rows) of the table. They are factored by Scale Factor (SF) to obtain the chosen database size. TPC-H provides a data generator called *dbgen*, which can be used to generate data for performance study.

Figure 6 contains eight tables, REGION, NATION, SUPPLIER, CUSTOMER, ORDERS, PART, PARTSUPPLIER and LINEITEM. All these tables are related through primary and foreign key attributes. Here nation and region tables are very small with only 5 and 25 rows, and require less storage space. For the performance study, it is not necessary to partition these table data. Instead, these tables can be replicated among all the nodes to improve the performance.

3.2 PROPOSED STORAGE STRUCTURES AND ITS ADVANTAGES

During the query execution, a query accesses data from different tables by performing join operation on the primary and foreign key attributes of those tables, and then the required attribute values are filtered. This becomes more complicated in the

virtual cloud environment, where the location of the data changes more often based on the resource availability or SLA (Service Level Agreement) [55] of the customer. Increased inter-node communication results in increased network delay and processing time, leading to the decrease in query performance. Hence, we need storage structures that help to minimize the communication between machines, minimize the maintenance during updates and minimize the increase in the storage space.

3.2.1 PK-MAP STRUCTURE

We create a PK-map (i.e., Primary Key map) for each of the primary keys in the tables of the star-schema of Figure 6. A PK-map will have one column for the primary key and one column for each of the foreign keys referencing this primary key as shown in Table 1. The primary key column contains primary key values and foreign key column contains logical record-ids. These logical record-ids are index to the Tuple-index-map of the foreign key. The logical record-ids run from 1 to n. The PK-maps are sorted on primary key values, which allow us to apply run length encoding on the 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 Chapter 5). As we know that the dimension tables are usually smaller in size and it stores the descriptive data, PK-map size will be small. Size of maps is analyzed in Chapter 5 and the experimental results are in Chapter 4.

Table 1: PK-map structure

Primary Key (PK) Column	Foreign Key1 (FK) Column	Foreign Key2 Column	.....
PK Value1	Starting index of Tuple-index-map1	Starting index of Tuple-index-map2	.....
PK Value2	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....

We will now apply the above PK-map structure to the TPC-H star-schema of Figure 6. For Figure 6, we need to create 7 PK-maps for keys, REGIONKEY, NATIONKEY, CUSTKEY, SUPPKEY, PARTKEY, PARTSUPPKEY (PARTKEY, SUPPKEY), and ORDERKEY. Table 2 shows the PK-map of REGIONKEY of the REGION table. The REGIONKEY is only referenced in the NATION table and hence table 2 has only 2 columns. The first column of table 2 stores the REGIONKEY attribute values and the second column stores the starting index of Tuple-index-map of the NATION table records/tuples that are mapping the REGIONKEY.

Similarly, Table 3 shows the PK-map of NATIONKEY of the NATION table. Here the NATIONKEY is referenced in two tables, CUSTOMER and SUPPLIER. Hence, table 3 has 3 columns. The first column stores the NATIONKEY attribute values of the NATION table. The second column stores the starting index of Tuple-index-map of the SUPPLIER table records/tuples that are mapping the NATIONKEY. The third column stores the starting index of Tuple-index-map of the CUSTOMER table records/tuples that are mapping NATIONKEY.

Table 2: RegionKey-map

	r_regionkey	n_rk_mindex
Row 0	0000000000	0
Row 1	0000000001	5
Row 2	0000000002	10
Row 3	0000000003	15
Row 4	0000000004	20

Table 3: NationKey-map

	n_nationkey	s_nk_mindex	c_nk_mindex
Row 0	0000000000	0	0
Row 1	0000000001	4	6
Row 2	0000000002	14	26
Row ..	.....	.....	.....
Row ..	.....	.....	.....
Row 24	0000000024	.....	.....

3.2.2 TUPLE-INDEX-MAP STRUCTURE

We create a Tuple-index-map for every foreign key that is referencing primary key of the PK-map of the star-schema (Figure 6). But, if the foreign key table is sorted on the foreign key attribute value, then we do not require a Tuple-index-map for that relationship. In that case, logical record-id of the PK-map will be the actual record-id of the foreign key table. For example, in Figure 6, PART and PARTSUPP tables are sorted on PARTKEY attribute. Hence, the PARTKEY-map does not require a Tuple-index-map. Similarly, ORDERS and LINEITEM tables are sorted on the ORDERKEY attribute and we don't need a Tuple-index-map for this relationship.

In the column-oriented databases, if even one projection of the table is sorted on the foreign key attribute, then the logical record-id of the PK-map will be the actual

foreign key record-id.

The Tuple-index-map will have only one column, which stores the actual record-ids of the foreign key table as shown in (Table 4). We create the Tuple-index-map for all those tables, which are not sorted on the foreign key attribute values. Thus, we create 6 Tuple-index-maps for the star-schema of Figure 6. One REGIONKEY-Tuple-index-map for the REGIONKEY in the NATION table, two NATIONKEY-Tuple-index-maps for the NATIONKEY in SUPPLIER and CUSTOMER tables, one SUPPKEY-Tuple-index-map for the SUPPKEY in PARTSUPP table, one CUSTKEY-Tuple-index-map for the CUSTKEY in ORDERS table, and one PARTSUPPKEY-Tuple-index map for the (PARTKEY, SUPPKEY) in LINEITEM table.

Table 4: RegionKey Tuple-index-map

	n_rk_tindex
Row 0	0
.	5
.	.....
Row 9	24
Row 10	8
.	9
.	.....
Row 19	23
Row 20	4
.	.....
Row 24	20

The Tuple-index-map will store the mapping between the logical and actual record-id of the foreign keys in the foreign key table as shown in Table 4. This is a REGIONKEY-Tuple-index-map created for the REGIONKEY in NATION table. Using

Table 4 we can extract the actual record-id mappings of the REGIONKEY of REGIONKEY-map (Table 2) and the NATION table. For example, the REGIONKEY "0000000000" has a value 0 in its second column and the key "0000000001" has a value 5. Thus, rows 0 to 4 in the REGIONKEY-Tuple-index map belongs to the key "0000000000". Records of the NATION table that has the REGIONKEY "0000000000" are those rows that are in rows 0 to 4. Similarly, other Tuple-index-maps are created. Since the local record-id of foreign keys is same as the map index, we don't store these values in the Tuple-index-map.

Both PK-maps and Tuple-index-maps are horizontally partitioned by the primary key attribute value and distributed to all the nodes containing the corresponding data. For example, REGIONKEY-map (Table 2) 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 the Tuple-index-map of the foreign keys is partitioned to correspond to the partitions in the PK-map. The REGIONKEY-Tuple-index-map (Table 4) 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. Replication of these maps can be done to improve the throughput or disaster recovery.

In the schema of Figure 6, 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 the 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.

3.2.3 ADVANTAGES OF PK-MAP AND TUPLE-INDEX-MAP

The first advantage is that, with the help of the primary key and foreign key relationship information stored on PK-maps and Tuple-index-maps, each node can execute a query independently 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 the supplier, nation and region tables of Figure 6. If these tables are stored among different nodes then we have to communicate with those nodes to get the mapping between these 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 the join between different tables. We will show the detailed processing of some of the TPC-H queries in the coming sections.

Secondly, PK-maps and Tuple-index-maps do not store any node specific information such as list of tables on neighboring databases, table's partition keys, and its attribute key ranges. By not storing such node specific information, we do not have to update meta-data every time the data is redistributed during scale-out or scale-in. This is important because, analytical applications run read-only queries on historical data and they update the data in batches after certain interval of time. And in cloud data warehouses, meta-data maintenance happens more often than data update. Thus, we sacrifice some node level query processing optimizations in order to gain the performance of meta-data maintenance. We use the broadcast method instead of a unicast

method to exchange the intermediate results of a query with nodes in the cluster.

3.2.4 REFERENCE GRAPH

Figure 7 is the reference graph for the TPC-H star-schema (Figure 6). This graph is a hierarchical representation of tables in the star-schema. It is used to put an order to the processing of tables in the query and filter as many unwanted data as possible in the early stages of the query processing.

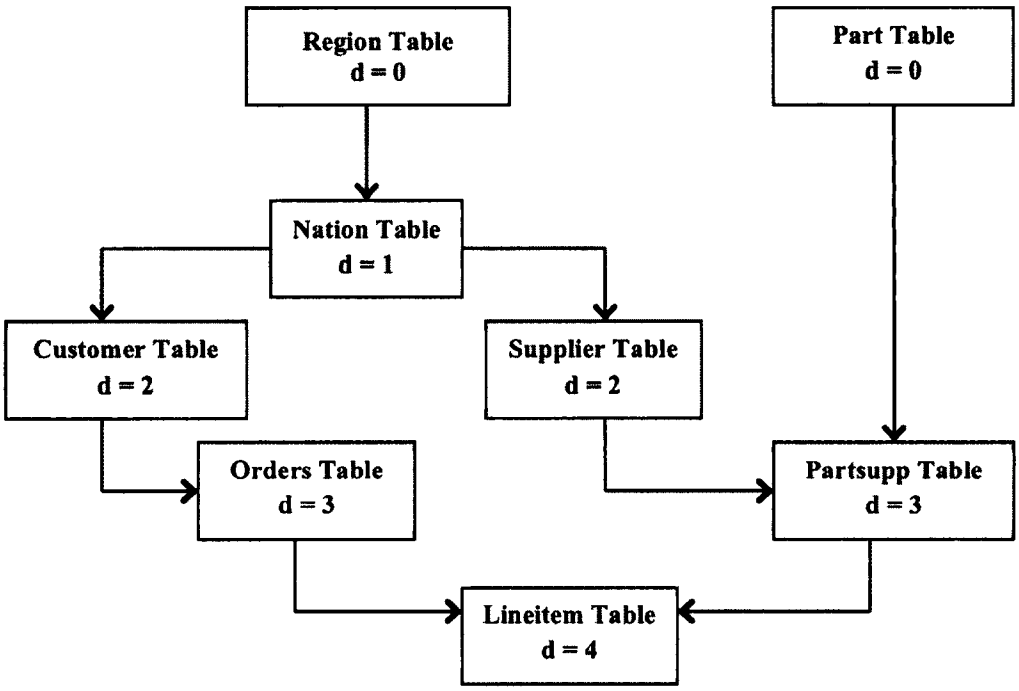


Figure 7: Reference Graph for star schema of Figure 6

Each rectangle box in Figure 7 represents a table of the star-schema and directed arrows connecting these boxes represent the relationship between the tables. For example, arrow connecting REGION table to NATION table means the primary key of

REGION table is referenced 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). "d=0" means there is no foreign key in the current table which refers to the primary key of other table. "d=1" means there are one or more foreign keys in the current table which refers to the primary key of other table with d=0. "d=2" means there are one or more foreign keys in the current table which refers to the primary key of other table with d=1, etc.

3.3 JOIN QUERY PROCESSING

In this Section, we analyze and optimize the join query processing in a highly distributed Cloud Data Warehouse, where each database stores a subset of relational data in a star-schema. In analytical query, join operations between two different tables are most common. To perform this join operation in a distributed architecture, data needs to be transferred among different nodes. This becomes critical when there is a huge amount of data (in Terabytes or Petabytes) stored across a large number of nodes. In addition, the metadata containing node specific data distribution information needs to be carefully redesigned and maintained in order to have elasticity in the cloud network.

With the increase in number of nodes and amount of data, the number of communications and message size also increase. This results in increased bandwidth usage and performance degradation. Thus, we design a query processing algorithm, which processes the join operation locally and independently with the help of proposed storage structures.

We will use Algorithm 1 shown below to process the multi-join query along with

the help of proposed PK-maps, Tuple-index-maps and a reference graph. Ad-hoc query and a reference graph are provided as input to the algorithm 1. High-level overview of this algorithm is that, 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.

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
      //d value is shown in Figure 2
3: for each table t 2 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 ids
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 t1with d1 _d referenced by t then
11:               Apply predicate on t
12:               Update the mapping with rec-ids of t
13:               Perform line 9
14:               Eliminate mappings which has no match for t
15:            else
16:               Perform similar to line 6, 9, and 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
```

For all those join predicates present in the query, we scan PK-map and Tuple-index-maps. Instead of communicating with the peer nodes and then performing the join operation, algorithm 1 on each node performs its local map scan and adds the required record-ids to the result. We store the result of applying predicates in the form (rec-id_1, rec-id_2, rec-id_3, ...), where rec-id_1 will be the record id of first table with d=0 and rec-id_2 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 $record_i$, then we eliminate the previously stored result (rec-id_1, rec-id_2,....., rec-id_{i-1}). We will broadcast the intermediate result to other nodes if necessary. 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. Let us look at the multi-join query processing by taking one of the queries of TPC-H benchmark in the below case study.

3.3.1 CASE STUDY: TPC-H QUERY PROCESSING

We chose 3 out of 22 TPC-H queries as shown in Figure 8, Figure 10 and Figure 11 for detailed analysis and performance study of our small-scale virtual network (Chapter 4.1). In this section we will see step-by-step processing of one of those queries (Figure 8) using our proposed storage structures (Chapter 3.2) and 'join' processing algorithm, Algorithm 1. We then analyze how the proposed approach reduces the number of communications while processing the multi-join query.

Example 1

```
select S.acctbal, S.name, N.name, P.partkey, P.mfgr, S.address, S.phone, S.comment
from PART P, SUPPLIER S, PARTSUPP PS, NATION N, REGION R
where P.partkey = PS.partkey
        and S.suppkey = PS.suppkey
        and P.size = 15 and P.type like '%BRASS'
        and S.nationkey = N.nationkey
        and N.regionkey = R.regionkey
        and R.name = 'EUROPE'
        and PS.supplycost = (select min(PS1.supplycost)
                from PARTSUPP PS1, SUPPLIER S1, NATION N1, REGION R1
                where P1.partkey = PS1.partkey
                and S1.suppkey = PS1.suppkey
                and S1.nationkey = N1.nationkey
                and N1.regionkey = R1.regionkey
                and R1.name = 'EUROPE')
order by S.acctbal desc, N.name, S.name, P.partkey;
```

Figure 8: TPC-H Query 2 (Source: [9], page 30)

Example 1 in Figure 8 is a Minimum Cost Supplier Query. This is a business query that finds, in a given region 'EUROPE' for each part of type 'BRASS' and size 15, the supplier who can supply it at minimum cost. If several suppliers in that region offer the desired part type and size at the same (minimum) cost, the query lists the parts from suppliers with the 100 highest account balances. For each supplier, the query lists the supplier's account balance, name and nation; the part's number and manufacturer; the supplier's address, phone number and comment information.

This query is processed as shown in the flow diagram of Figure 9. This query involves five tables – REGION, NATION, SUPPLIER, PART, PARTSUPP; four join operations between those tables; and three filtering predicates on the region name (R.name), part type (P.type) and part size (P.size).

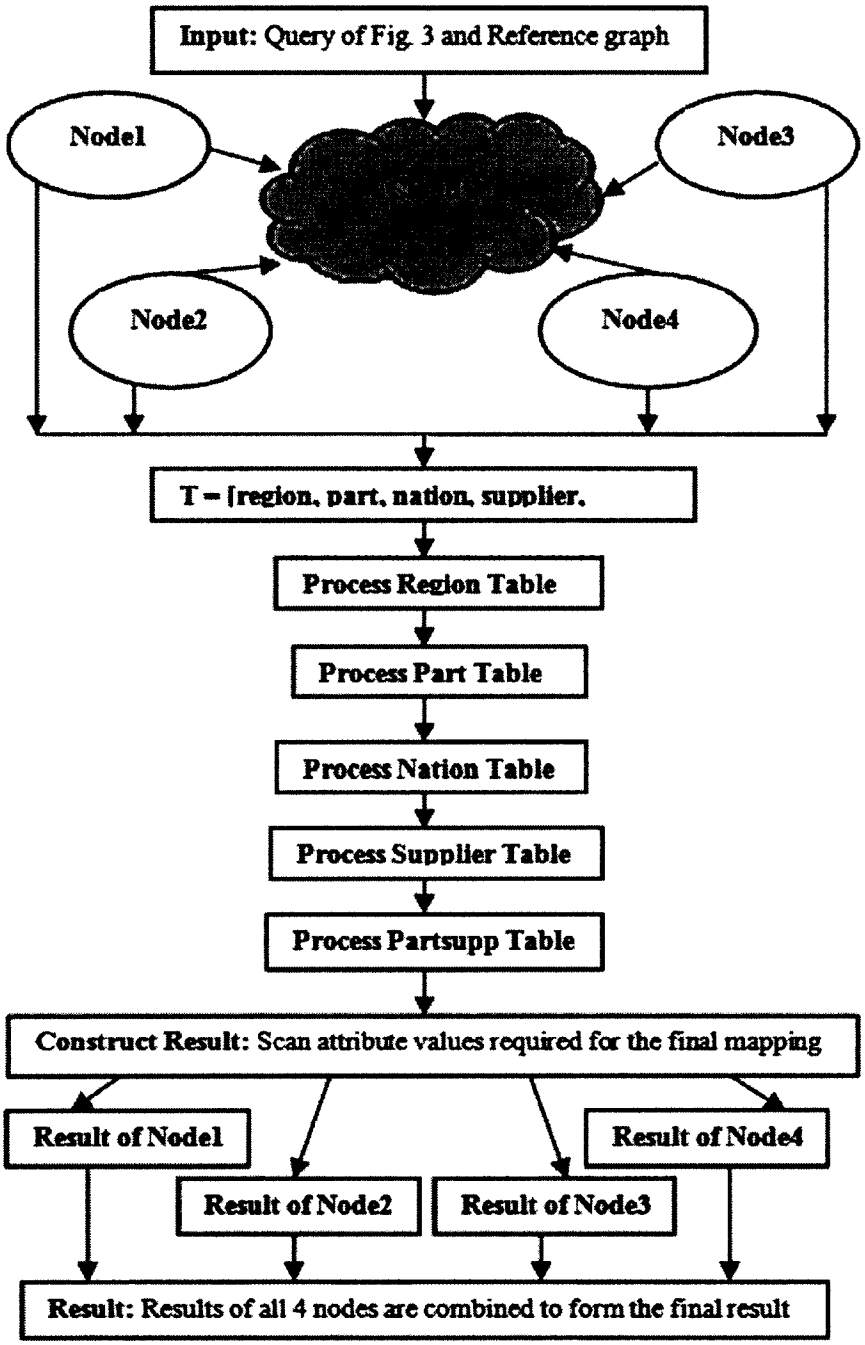


Figure 9: Flow diagram for processing Example 1

While generating the query plan we rewrite those queries that contain sub queries and 'having' clause into simple query with join and where clause. This can be done

because the sub queries can be converted into 'join' operation and the 'having' clause to 'where' clause.

Algorithm1 performs following steps in processing Example 1 query of Figure 8:

Input: Example 1 query of Figure 8, Reference graph of Figure 7

Sort: Tables in the 'from' clause of the query are sorted using reference graph

T = [REGION, PART, NATION, SUPPLIER, PARTSUPP]

Loop1: Each node will process the region table: (executes line 4 to 9 of Algorithm 1)

Scan the region table to get the region key with region name 'EUROPE' and broadcast the record-id to all other nodes.

For example, If node 1 finds the region key of EUROPE as "0000000003" and its PK-map record-id is 3, then node 1 will send record-id 3 to nodes 2, 3 and 4.

Loop2: Each node will process the part table: (Executes line 4 to 9 of Algorithm 1)

Scan the part table to get the part keys whose P.size=15 and P.type like '%BRASS'. Then, broadcast the record-ids of the result to every other node.

Loop3: Each node will process the nation table: (executes line 22 to 25 of Algorithm 1)

Scan REGIONKEY-map to get the record-ids of the region key in the nation table.

Loop4: Each node will process the supplier table: (execute line 22 to 25 of Algorithm 1)

Scan NATIONKEY-map to get the logical record-ids of nation key attribute in the supplier table.

Loop5: Each node will process the partsupp table:

Considering the results from Loop 2 and Loop 4, scan SUPPKEY-map and PARTKEY-map to find the record-ids that are mapped to the partsupp table.

Then, find the record-ids that are common to both the results.

For these common record-ids, find the minimum PS.supplycost. Broadcast the local minimum PS.supplycost with all the other nodes to find the global minimum PS.supplycost (line 11 to 14 in algorithm 1). With these results we find the final record-ids.

Final: Select all the attribute values required for the final result using the final mapping of the Loop5.

Partial results of each of the nodes of the network are then combined to form the final result.

In the Example 1 query processing we need only three communications between the nodes as shown in Loops 1, 2 and 5. But in the general approach, each join of two tables requires communication with other nodes. This is because a node does not know whether there is any record in other node that matches the current node's record. So communication is needed between the nodes after each of the operations like, find the region key for R.name 'EUROPE', join those results with the nation table, then join the results with the supplier table, find the part keys of the part table whose P.size is 15 and P.type like '%BRASS', join the results with the partsupp table to find the local minimum PS.supplycost and then find the final records.

In total, five communications are needed between all the participating nodes. So it is clear that using PK-map structure reduces the communication among the nodes during query processing. Also, in the general approach we need to exchange join attribute values instead of the join indices. The join attribute values can be of any type such as the variable length string that are larger in size than the record-ids. Similarly, we can process

Example 2 of Figure 10 and Example 3 of Figure 11 using the Algorithm 1.

Example 2

```
select N.name, sum (L.extendedprice *(1 – L.discount)) as revenue
from CUSTOMER C,ORDERS O, LINEITEM L, SUPPLIER S, NATION N, REGION
R
where C.custkey = O.custkey
        and L.orderkey = O.orderkey
        and L.suppkey = S.suppkey
        and C.nationkey = S.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;
```

Figure 10: TPC-H Query 5 (Source: [9], page 37)

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 C, ORDERS O, LINEITEM L, NATION N
where C.custkey = O.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;
```

Figure 11: TPC-H Query 10 (Source: [9], page 46)

In Chapter 4 we show through extensive experiments using a small-scale four-node virtual network and a large-scale PlanetLab Cloud network that, our proposed

storage structures PK-map and Tuple-index-map, and the join query execution algorithm improve the performance of multiple join queries, decrease the node-to-node communications and decrease the size of messages exchanged in Cloud Data Warehouses.

3.4 AGGREGATE QUERY PROCESSING

In this Section, we study and optimize the aggregate query processing in a highly distributed Cloud Data Warehouse, where each database stores a subset of relational data in a star-schema. Existing aggregate query processing algorithms focus on optimizing various query operations but give less importance to the communication cost overhead (Two-phase algorithm). However, in cloud architectures, the communication cost overhead is an important factor in the query processing. Thus, we consider communication overhead to improve the distributed query processing in such cloud data warehouses.

The aggregate operation is common in analytical queries. During the query execution, these aggregate operations are applied after grouping records on all attributes present in the group-by clause. Attributes in the group-by clause might be from different tables and hence, a join operation has to be executed before group-by. So, we use our proposed map structures to push down the group-by operation and reduce the cost of executing such queries.

We can optimize aggregate queries in different ways based on the type of aggregate attribute in the select clause or based on the attributes in group-by clause. Thus, in this Section we first categorize the aggregate queries. We then design a query-

processing algorithm by analyzing the categories of aggregate operation and eliminating most of the sort and group-by operations with the help of integrity constraints and our proposed storage structures, PK-map and Tuple-index-map. Extensive experiments on PlanetLab cloud machines validate the effectiveness of our proposed framework in improving the response time, reducing the node-to-node interdependency, minimizing the communication overhead, and reducing the database table access required for an aggregate query. Experiment results can be found in Chapter 4.2.

3.4.1 AGGREGATE OPERATIONS

Inferring on referential integrity constraints and functional dependencies, we have classified the aggregate operations into two general categories based on the type of attribute on which the aggregate operation is applied. We then generate a plan for query processing accordingly. We do not consider "having" clause in our analysis, because having clause can be converted into "where" clause constraints by rewriting the query [45].

3.4.1.1 AGGREGATE ON PRIMARY OR FOREIGN KEY

When there is an aggregate operation on the primary key (PK) or foreign key (FK), we do not scan the database table (unless there is a filtering constraint on non-PK or non-FK), instead we scan our proposed map structures and perform aggregate operation on the fly. We can do this because, our maps are sorted on keys, and they contain required information on the number of tuples of the foreign key table that is mapped to the primary key of dimension table.

For example, Query 1 of Table 6 has a count (*) operation. This is a business query trying to find the total number of orders placed by the specified nations. Here, we only require the number of rows in orders table that belong to each group in the group-by clause (i.e., N.name). By inference, we can say that, this is an aggregate operation on primary key (PK) of the orders table. Hence, we can eliminate the scan of orders table and get this count by scanning the CUSTKEY-map, which contains mapping information between the CUSTKEY of CUSTOMER table and the CUSTKEY of ORDERS table. We push down the group-by on N.name to step "scan NATIONKEY-map" as shown in Figure 12.

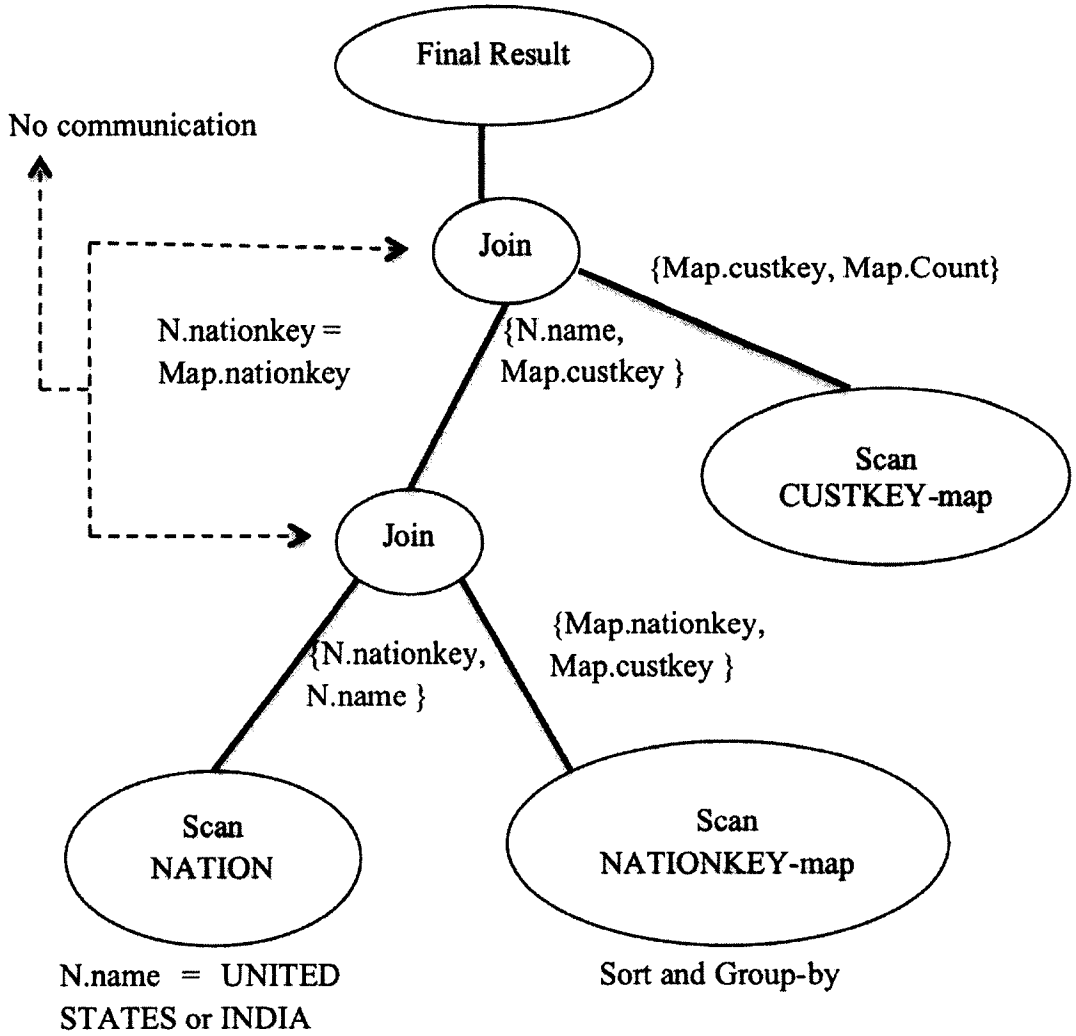


Figure 12: PK-map Query Plan for Query1 of Table 6

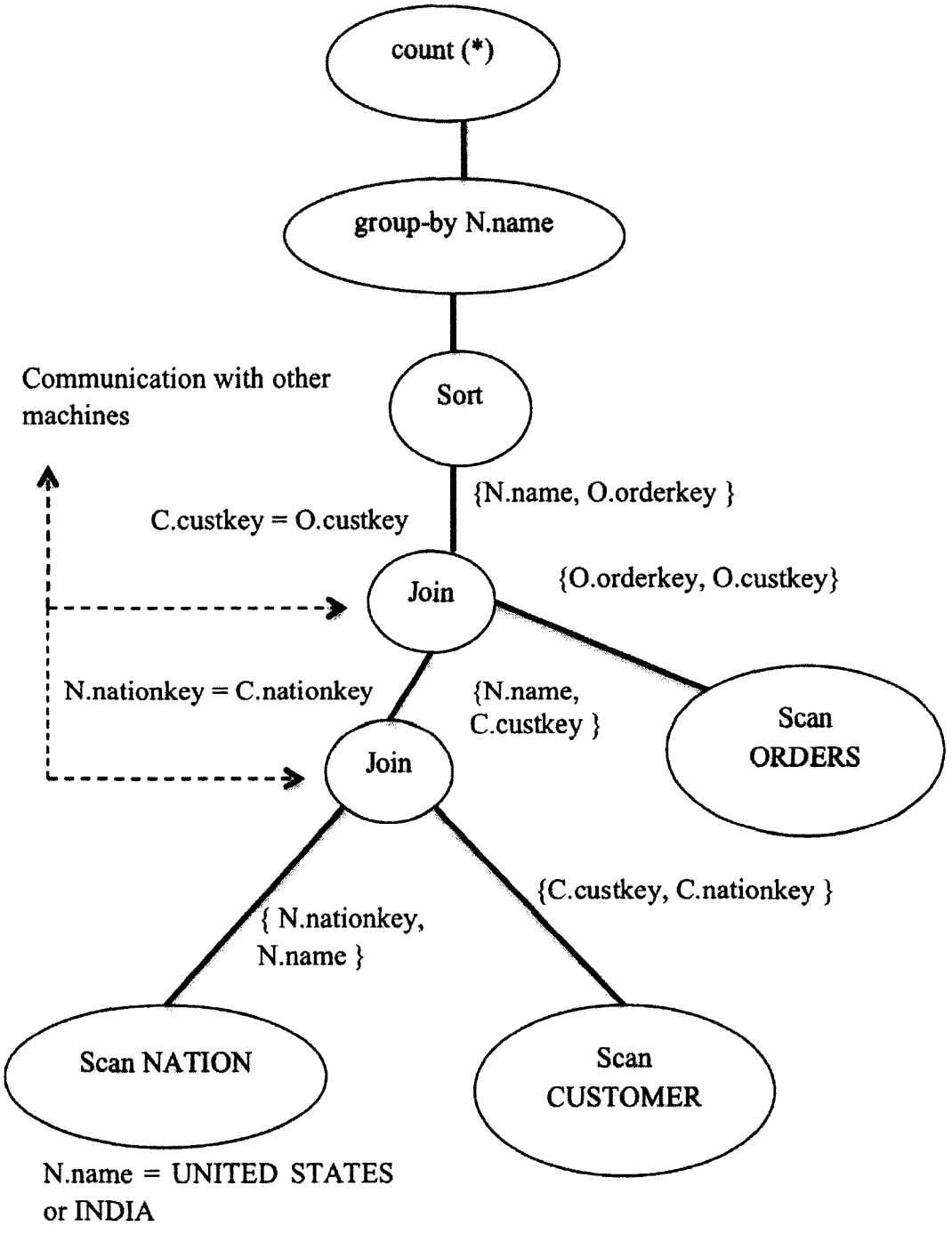


Figure 13: MySQL Query Plan for Query1 of Table 6

Figure 13 is the plan generated by MySQL for processing Query 1 of Table 6. As shown in Figure 13, it requires three tables scan (Nation, Customer and Orders), two

joins, one sort and a group-by operation. Since these tables are fragmented and distributed into different nodes, the query processor needs to communicate the data for each join operation. This increases the communication overhead and response time.

On the other hand, Figure 12 is the plan generated for the same Query 1 using our framework. Here, instead of scanning the Customer and Orders table, we scan two PK-maps (NATIONKEY-map and CUSTKEY-map), and its associated Tuple-index-maps. These maps are very small compared to scanning tables. The join operation in Figure 12 is not equivalent to the join operation in Figure 13. Instead, it is performed by scanning the map and applying associated filtering constraints. Algorithms are shown in Algorithm 2 and Algorithm 1 and comparison results in Chapter 4.2.

3.4.1.2 AGGREGATE ON NON-PRIMARY OR NON-FOREIGN KEY

When there is an aggregate operation on Non-Primary Key (NPK) or Non-Foreign Key (NFK) of the table, then we need to scan the table to find the correct result. But, we use inference to push down the group-by in the query tree so that we can scan only the required portion of the table. We also replace the scanning table with maps wherever possible.

For example, suppose we have a SUM (O.totalprice) instead of a COUNT (*) in Query 1 as shown in Figure 14. We push down the group-by of N.name (as done in Figure 12) to reduce the number of rows to be scanned from the orders table. This reduces the input rows of the final aggregate operation.

Similarly, if we have an aggregate operation AVG, we push down the count operation along with the group-by. Transformations are applied for duplicate elimination

queries in order to retain the relevant data.

```
select N.name, SUM(O.totalprice)
from ORDERS O, CUSTOMER C, NATION N
where N.name = "UNITED STATES"
        or N.name = "INDIA"
        and N.nationkey = C.nationkey
        and C.custkey = O.custkey
group by N.name;
```

Figure 14: Modified Query 1 of Table 6

3.4.2 GROUP-BY OPTIMIZATION

When there are multiple attributes in the group-by clause, we associate certain order to those attributes and push down the group-by in the query tree. As we know that the group-by is a set operation and join result will carry the sort order, we can change the sequence of attributes in the group-by clause. So, we change the sequence of attributes in the group-by clause corresponding to the sequence of table processing in our algorithm. By doing this we can eliminate non-relevant data in the early stage of the query processing as well as achieve same result as we perform group-by in the final stage.

3.4.3 QUERY PROCESSING USING PROPOSED METHOD

We use both the Algorithm 1 and Algorithm 2 to process the aggregate query. Algorithm 1 and Algorithm 2 show a general approach of aggregate query processing. Mainly, we consider the aggregate operation to decide whether to scan the table from the database or to scan our map structure (Line 3-15, Algorithm 2).

Algorithm 2 Aggregate Query Processing Algorithm

Input: query Q, reference graph G (Figure 3)
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 and filtering constraints
       //d value is shown in reference graph of Figure 3
 3: for each table t ε T do
 4:     Perform predicate/join processing using Algorithm 3
 5:     if there is an aggregate on PK or FK then
 6:         Scan PK-map
 7:         //sort and group-by is inferred
 8:         Update the current result set
 9:     else if there is an aggregate on NPK or NFK then
10:         Scan t and apply aggregate operation
11:       //sort and group-by is applied if necessary
12:        Update the current result set
13:        Communicate if necessary with other machines
14:    end if
15: end for
16: Scan tables of T if necessary to get the value of other attributes
    in the select statement of Q
17: return Result
```

If there is a group-by on the primary key (PK) or foreign key (FK) attribute, we scan our maps. In this case, we eliminate the communication and data movement because all the required mapping information on PK to FK mapping is available in our maps (Lines 5-8 Algorithm 2). Also, our maps are already sorted on keys, which further eliminate most of the sort operations.

If there is a group-by on the Non-PK or Non-FK attribute, we need to scan the table data. But, our maps are already sorted on keys, which allow us to eliminate most of the sort operations. If possible, we push down the group-by operation along with the dimension table scan to reduce the cost of final aggregate operation. We use the

Algorithm 1 for each join operation in the query. At last we retrieve remaining attributes required for the result.

Algorithm 1 is the same algorithm we used in Section 3.3 for join query processing. In Algorithm 1, we first sort the tables referenced in the query according to their relationship in the schema. We sort tables starting from the table that does not have any foreign keys (depth 0). For example, Query 1's order of processing will be nation --> customer --> orders. In addition, we consider all the constraints in *where* clause and *group-by* clause while deciding on tables processing order. Algorithm 1 is explained in more detail with the case study in Section 3.3.

Table 5: Queries for evaluating Join Performance

Query 1: *Find the suppliers from 'EUROPE', who can supply given part type, and size at a minimum supply cost.*

```
select S.name, PS.supplycost
from SUPPLIER S, PARTSUPP PS
where S.suppkey = PS.suppkey and PS.supplycost = (select min (PS1.supplycost)
        from NATION N, REGION R, SUPPLIER S1, PART P, PARTSUPP PS1
        where R.name = 'europe' and P.size = 15 and P.type like '%brass'
                and R.regionkey = N.regionkey and N.nationkey = S1.nationkey
                and S1.suppkey = PS1.suppkey and P.partkey = PS1.partkey)
```

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

```
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.*

```
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*

```
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*

```
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 PS.availqty > 9900
        and N.nationkey = S.nationkey and S.suppkey = PS.suppkey
        and P.partkey = PS.partkey;
```

Table 6: Queries for evaluating Aggregate Performance

Query 1: *(COUNT) Find the total number of orders placed by the 'UNITED STATES' and 'INDIA' customers (aggregation on primary key attribute)* select N.name, count(*) from ORDERS O, CUSTOMER C, NATION N where N.name = "UNITED STATES" or N.name = "INDIA" and N.nationkey = C.nationkey and C.custkey = O.custkey group by N.name;
Query2: *(COUNT) Find the total number of failed orders placed by customers of each nation. I.e, orders whose ORDERSTATUS = 'F' (aggregation on NON – Primary Key attribute)* select N.name, count(*) from ORDERS O, CUSTOMER C, NATION N where N.nationkey = C.nationkey and C.custkey = O.custkey and O.orderstatus = 'F' group by N.name;
Query 3: *(MIN) Find the minimum supply cost for all the parts supplied by 'UNITED STATES' suppliers* select PS.partkey, min(PS.supplycost) from PARTSUPP PS, SUPPLIER S, NATION N where N.name = "UNITED STATES" and N.nationkey = S.nationkey and S.suppkey = PS.suppkey group by PS.partkey;
Query 4: *(SUM) Find the revenue generated by customers of each nation in the year 1995. I.e., revenue is equal to the totalprice from the orders table* select N.name, sum(O.totalprice) from ORDERS O, CUSTOMER C, NATION N where N.nationkey = C.nationkey and C.custkey = O.custkey and O.orderdate like '1995% group by N.name;
Query5: *(AVG) Find the average revenue generated by customers of each nation in year 1995. i.e., revenue is equal to the totalprice from the orders table* select N.name, avg(O.totalprice) from ORDERS O, CUSTOMER C, NATION N where N.nationkey = C.nationkey and C.custkey = O.custkey and O.orderdate like '1995% group by N.name;

4. PERFORMANCE EVALUATION

4.1 JOIN QUERY PERFORMANCE RESULTS

In this section, we present the performance study to evaluate the effectiveness of our proposed PK-map and Tuple-index-map structures using the Algorithm 1. First, we will analyze the performance on a small-scale cluster of four virtual nodes. Second, we will study the performance on a large-scale cluster called "PlanetLab" with fifty nodes and 150GB of data.

4.1.1 SMALL-SCALE NETWORK

We chose three out of 22 TPC-H queries as shown in Figure 8, Figure 10 and Figure 11 for performance study of the small-scale network. Here, we measure the performance of average query execution time, number of communications made by the 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 a 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) [25] [34] [35].

To perform experiments, we generated data using the data generator *dbgen*

provided by the TPC-H benchmark and distributed it to all four nodes. We generated 10GB of data for the experiments in the Sections 4.1.1.2 to 4.1.1.4 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 the same partition keys as data partition. We took same three queries, which we have analyzed in Chapter 3.3 to analyze the performance.

4.1.1.1 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 the *dbgen* and then created PK-maps and Tuple-index-maps for each of these data. Size of these maps is shown in Figure 15.

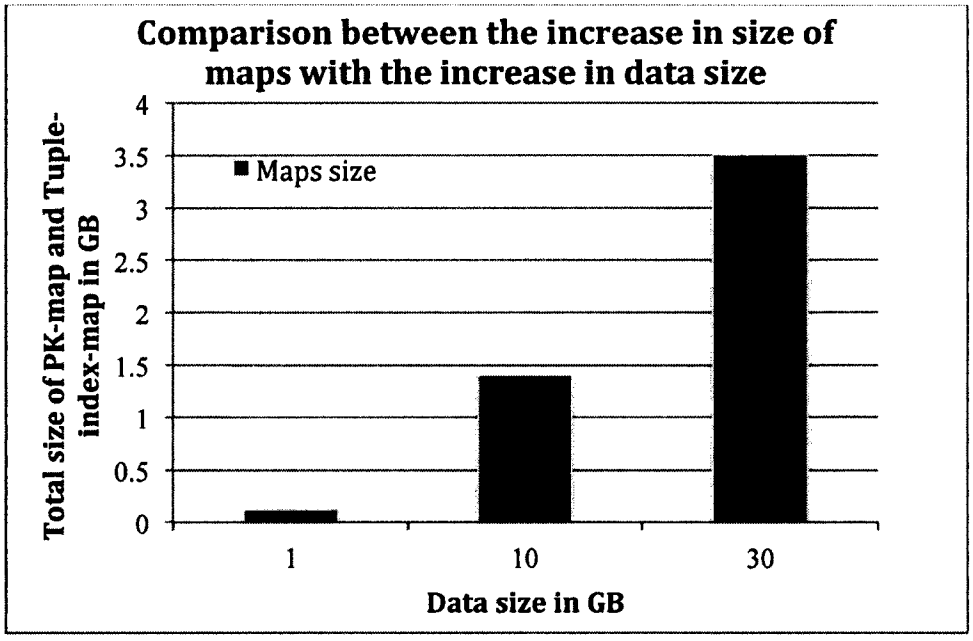


Figure 15: Comparison between the increase in size of maps with the increase in size of data

We can clearly observe from Figure 15 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.

4.1.1.2 SIZE OF INTER-NODE COMMUNICATION MESSAGES

To compare the inter-node communication message size between the proposed approach 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 Figure 16.

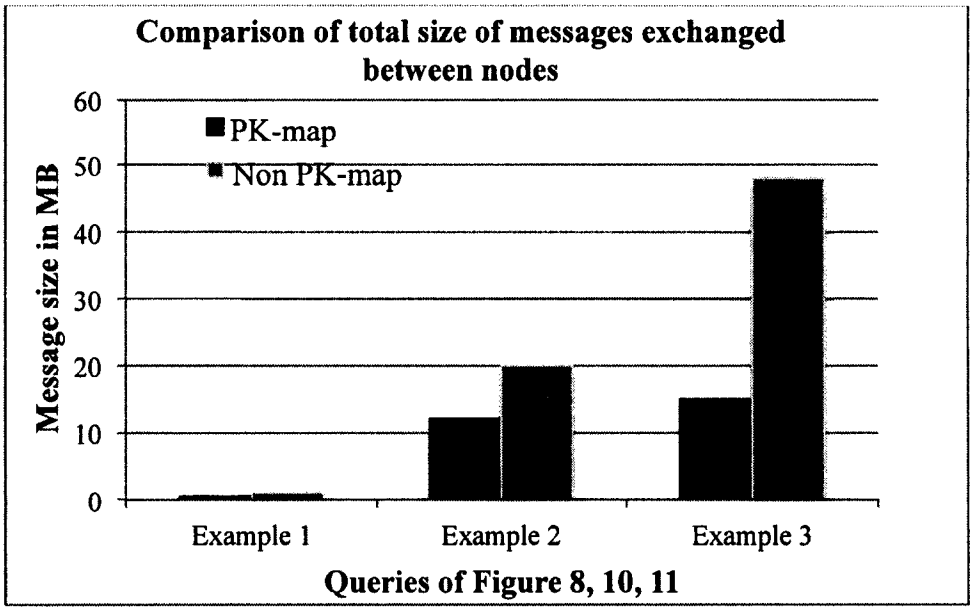


Figure 16: Comparison of total size of messages exchanged between nodes during query execution

In Figure 16, 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 exchanged in our approach and the second bar (in blue) shows the total size of the messages exchanged in general approach.

As we can see in the graph (Figure 16), execution of Example 1 query exchange 0.5MB of data in our approach and 1.1MB of data in the general approach. Execution of Example 2 query involves exchange of 12MB of data in our approach and 20MB in the general approach. Execution of Example 3 query involves exchange of 15 MB of data in our approach and 48MB in the general approach.

Based on the result in Figure 16, 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 the actual join attribute values.

4.1.1.3 AVERAGE QUERY EXECUTION TIME

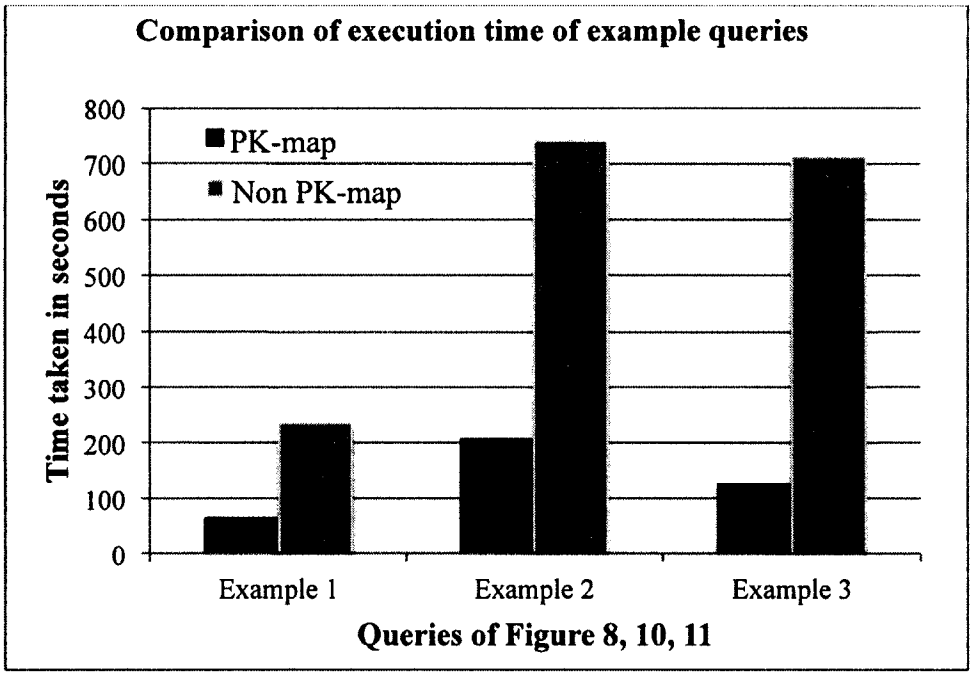


Figure 17: Comparison of time taken for query execution

Comparison of average time taken by the queries in Figure 8, Figure 10 and Figure 11 are shown in Figure 17. In this graph (Figure 17), 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 state-of-the-art approach.

As we can see from this graph (Figure 17), Example 1 query takes 63 seconds to run in our approach and 235 seconds in the general approach. Example 2 query takes 206 seconds in our approach and 741 seconds in the general approach. Example 3 query takes 125 seconds in our approach and 712 seconds in 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 Figure 17, it is clear that, the query execution time required by the state-of-the-art approach is more than our approach.

4.1.1.4 NUMBER OF INTER-NODE COMMUNICATION

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

Comparisons on the total number of messages exchanged during the execution of queries are shown in Figure 18. In this graph (Figure 18), 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.

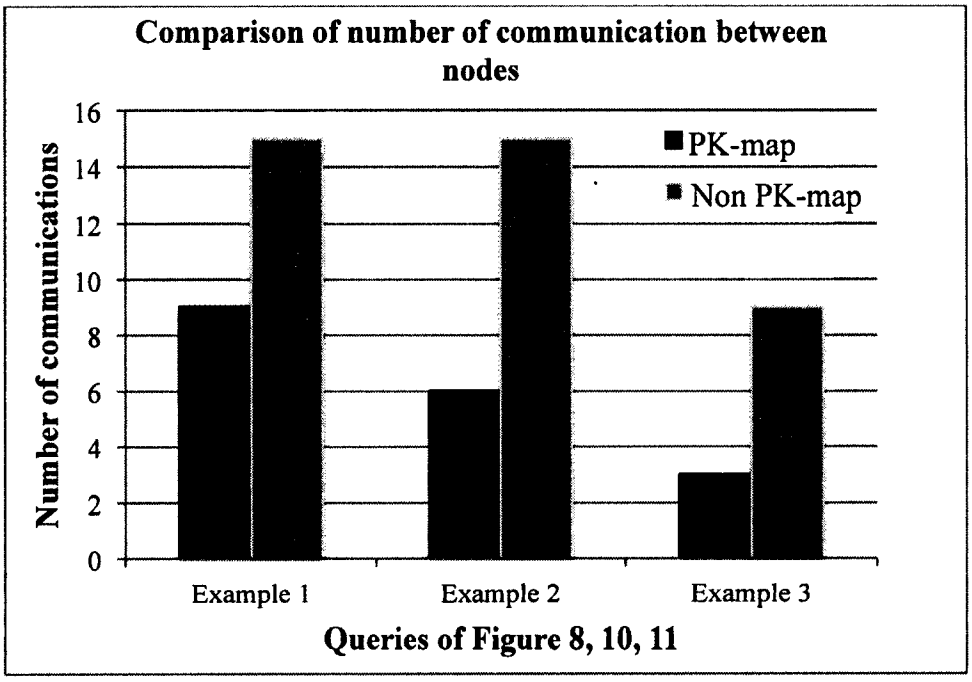


Figure 18: Comparison of number of communications between nodes executing query

Based on the result in Figure 18, 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.

4.1.2 LARGE-SCALE CLOUD NETWORK

We used PlanetLab network for the performance study of this Section. **PlanetLab** [37] is a group of computers available as a test bed for researchers to develop, deploy and test their applications. It is currently composed of around 1050 nodes (servers) at 400 sites (location) worldwide.

For experiments we used 50 PlanetLab machines located at different parts of the world, running a Red Hat 4.1 Operating System. Each machine has a 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 the 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) [25] [34] [35].

To perform experiments, we generated 150GB of data using *dbgen* provided by the 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 5.

4.1.2.1. QUERY EXECUTION TIME

Comparison of time taken by the queries in Table 5 is shown in Figure 19 to Figure 23. In the graphs of Figure 19 to 23, 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 state-of-the-art approach.

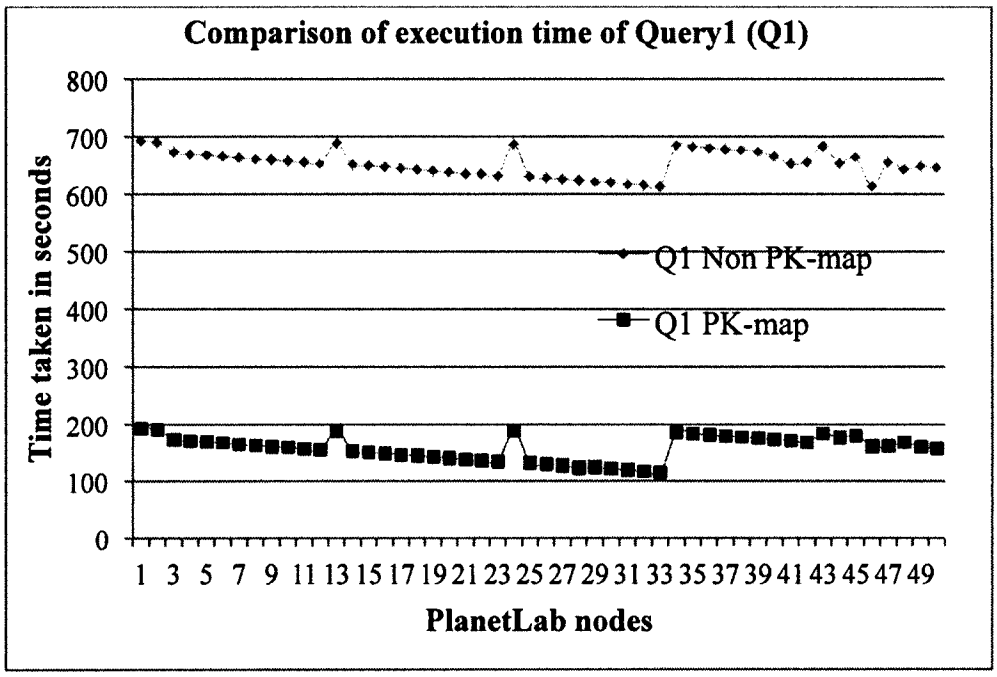


Figure 19: Comparison of time taken for executing Query1 of Table 5

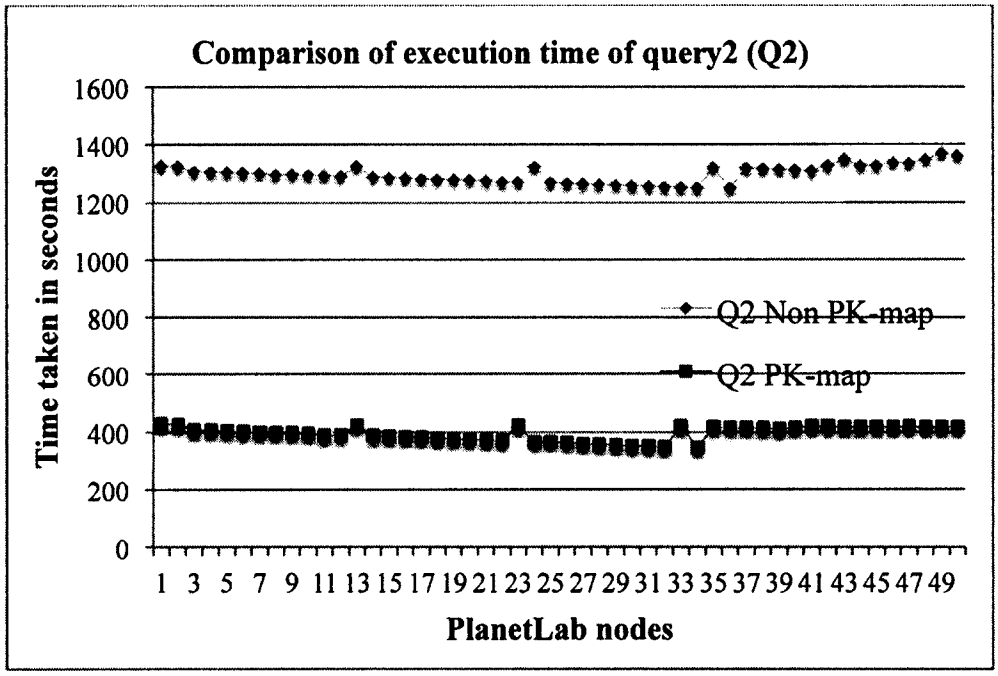


Figure 20: Comparison of time taken for executing Query2 of Table 5

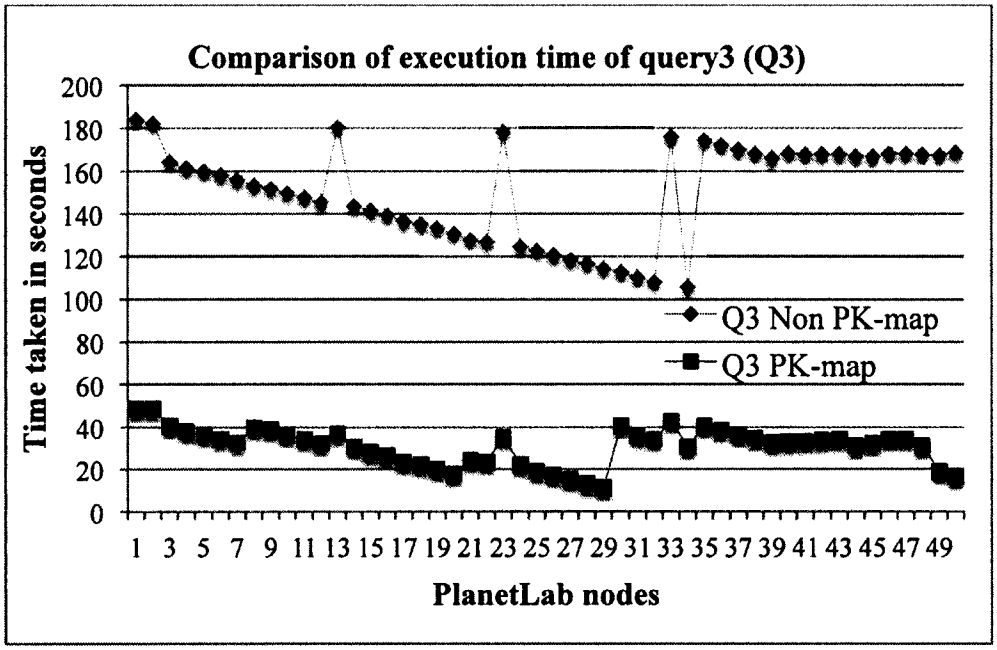


Figure 21: Comparison of time taken for executing Query3 of Table 5

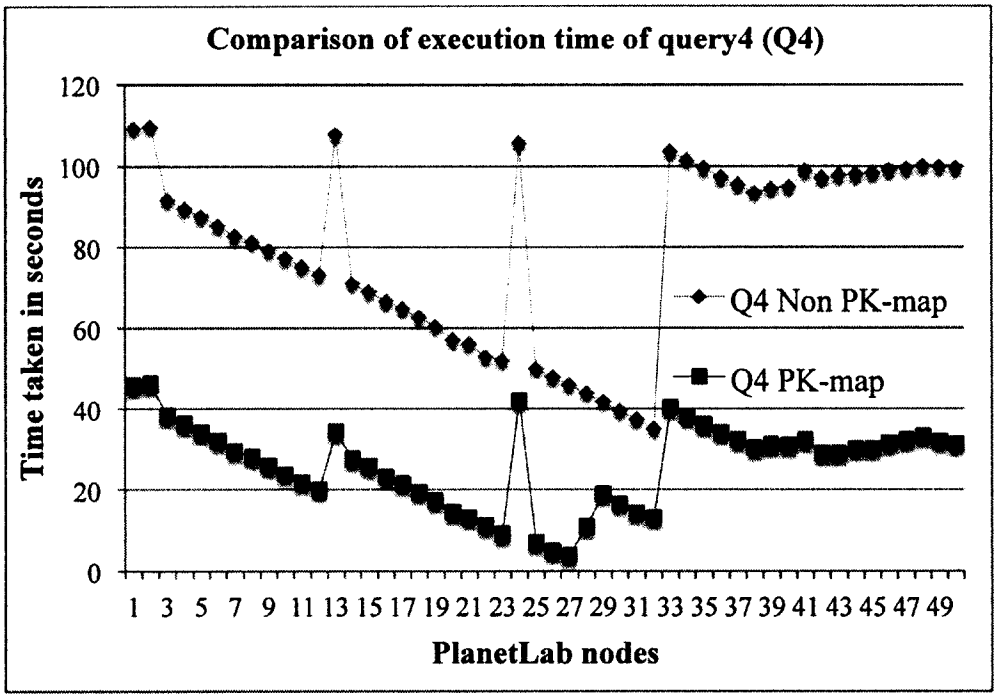


Figure 22: Comparison of time taken for executing Query4 of Table 5

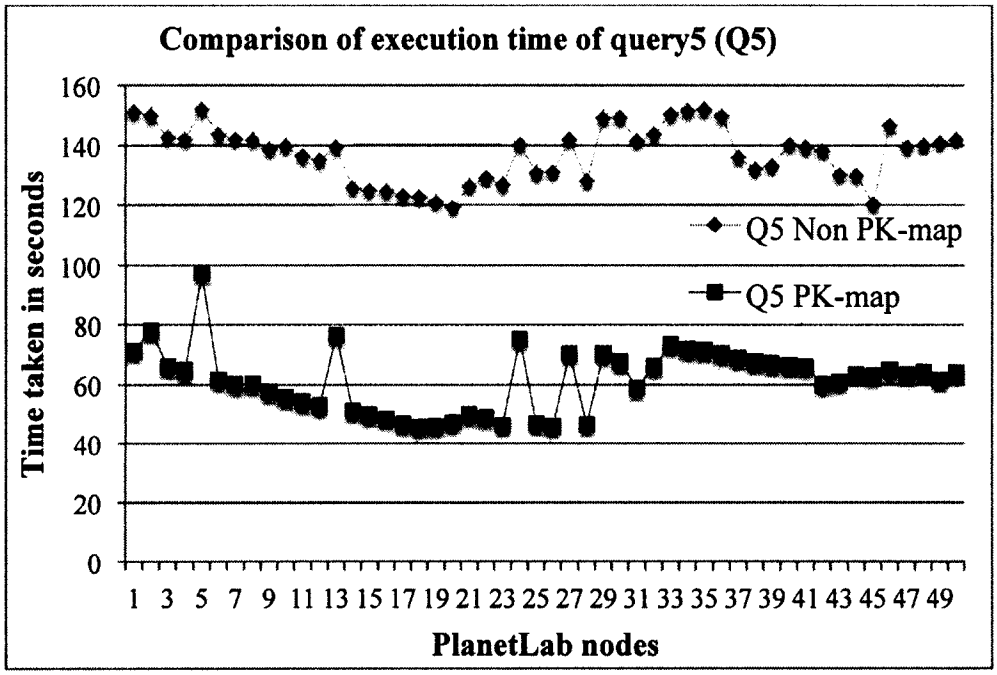


Figure 23: Comparison of time taken for executing Query5 of Table 5

Each node has to communicate the 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 the 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 Figure 19 to 23, it is clear that, the query execution time required by the state-of-the-art approach is more than our approach.

4.1.2.2. TOTAL NUMBER OF INTER-NODE COMMUNICATION

As explained in the previous Section 4.1.2.1, each inter-node communication has

some inherent overhead. Thus, we also compare the number of messages exchanged in addition to the total message size.

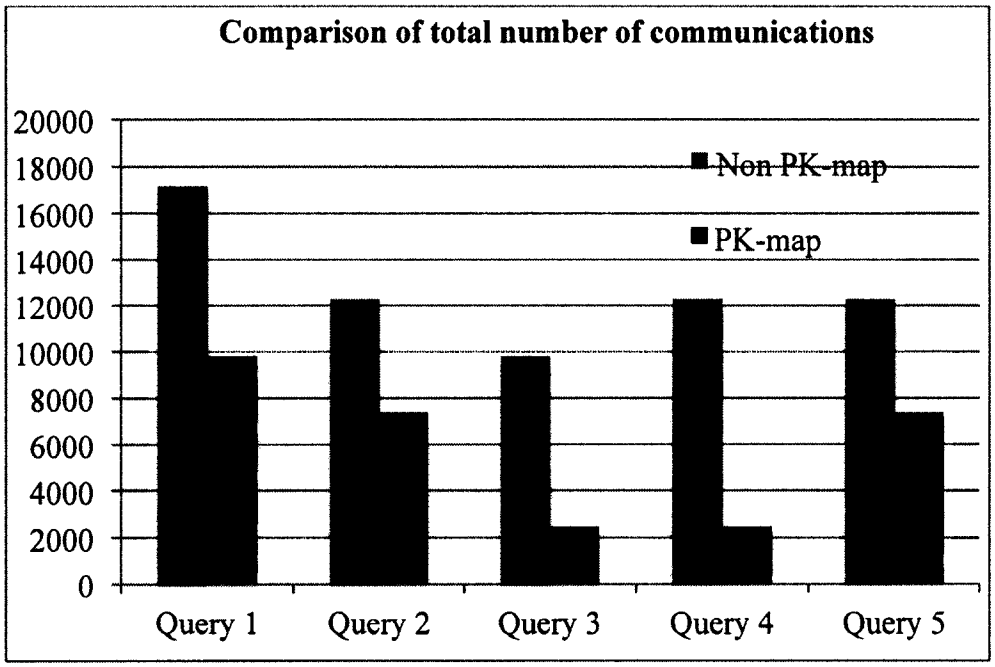


Figure 24: Comparison of total number of communications for executing Table 5 queries

Comparisons on the total number of messages exchanged during the execution of queries are shown in Figure 24. In this graph (Figure 24), 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.

In this thesis we were unable to run tests using real data set and on a commercial cloud. Since we were able to achieve significant performance improvement using PlanetLab, we can say that the experiments are proofs of concept on the saving of

communication costs for query execution on any commercial cloud. In a large commercial cloud environment where the network often changes due to the resource availability and SLA [55], the communication cost has even greater weight in the query costs [23] [24]. Therefore, the savings resulted from our work would be even more significant.

4.2 AGGREGATE QUERY PERFORMANCE RESULTS

In this section, we present the performance study to show the effectiveness of our proposed PK-map and Tuple-index-map structures while processing aggregate queries (using Algorithms 1 and 2). We will compare the performance between MySQL and our proposed framework on a large-scale cloud network called PlanetLab with 150GB of TPC-H star-schema data.

PlanetLab [36] [37] is a geographically distributed computing platform available as a testbed for deploying, evaluating, and accessing planetary-scale network services. It is currently composed of around 1050 nodes (servers) at 400 sites (location) worldwide.

For performance study of this Section, we chose 50 PlanetLab machines worldwide running a Red Hat 4.1 Operating System. Each machine has the 2.33GHz Intel Core 2 Duo processor, 4GB RAM and 10GB disk space. We installed regular MySQL on all of the machines to perform experiments. We generated 150GB of data using the data generator *dbgen*, provided by the TPC-H benchmark and distributed it to 50 PlanetLab machines. Each of these machines store around 3GB data fragments of the TPC-H schema relations.

We generated PK-maps and Tuple-index-maps, and then horizontally partitioned

them using the same partition key that was used to partition the data. We then distributed these partitions into all 50 machines that contained corresponding data. All the map structures are loaded into the main memory before each experiment begins. Here we can take advantage of the main memory databases if the data size is huge (in Petabyte or Exabyte) [25] [34] [35].

We used 5 queries for the performance analysis as shown in Table 6. Comparison of time taken by these queries is shown in Figure 25 to 29. In the graphs of these figures, PlanetLab machines are on the x-axis and time taken by the query (in seconds) is on the y-axis. On each graph, top blue line shows the result of processing aggregate queries on MySQL, and the bottom red line shows the result of our framework. For simplicity of explanation, we do not consider pipelined approach to count the number of communications between the machines. Instead we use the inter query operation communications.

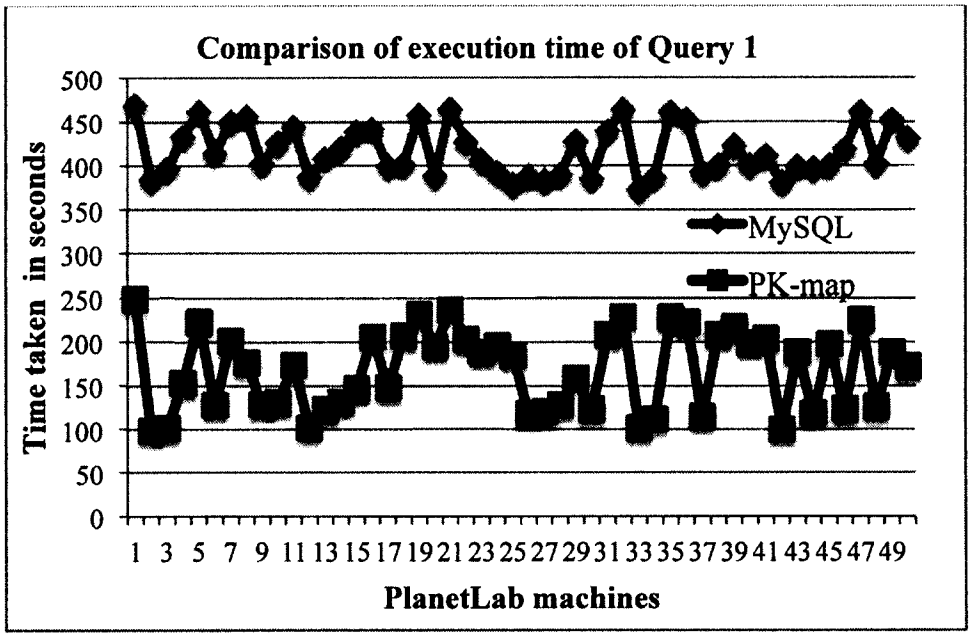


Figure 25: Performance Comparison of Query 1 of Table 6

Figure 25 compares the time taken for processing Query 1 of Table 6. As explained in the Chapter 3.4.1, Query 1 has an aggregate operation on the primary key of the table ORDERS, and a Non-Primary Key/Non-Foreign Key constraint on the table NATION. Hence, it is sufficient to scan the NATION table, NATIONKEY-map, and CUSTKEY-map to answer this query.

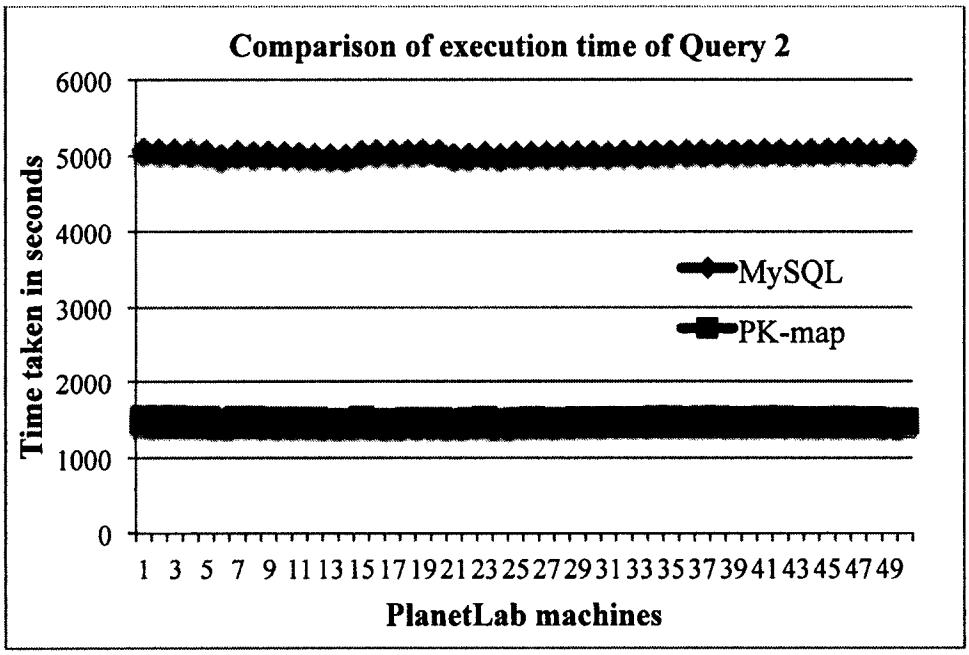


Figure 26: Performance Comparison of Query 2 of Table 6

Figure 26 compares the time taken for processing Query 2 of Table 6. In Query 2, we have an aggregate operation COUNT (*), and a Non-Primary Key/Non-Foreign Key constraint on the attribute O.orderstatus of table ORDERS. Hence, we need to scan the ORDERS table along with the NATIONKEY-map and CUSTKEY-map to process this query.

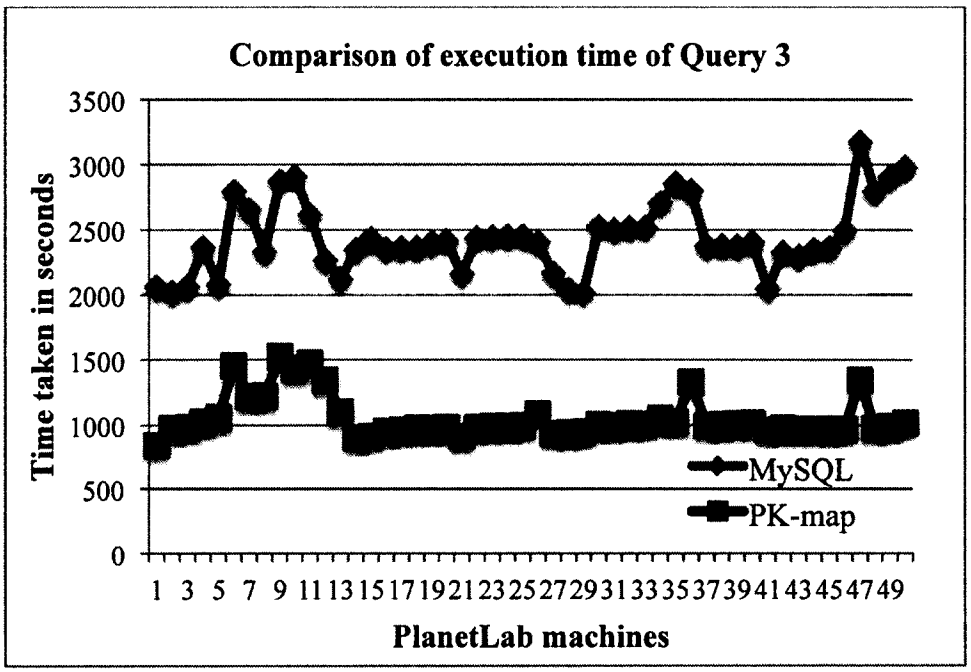


Figure 27: Performance Comparison of Query 3 of Table 6

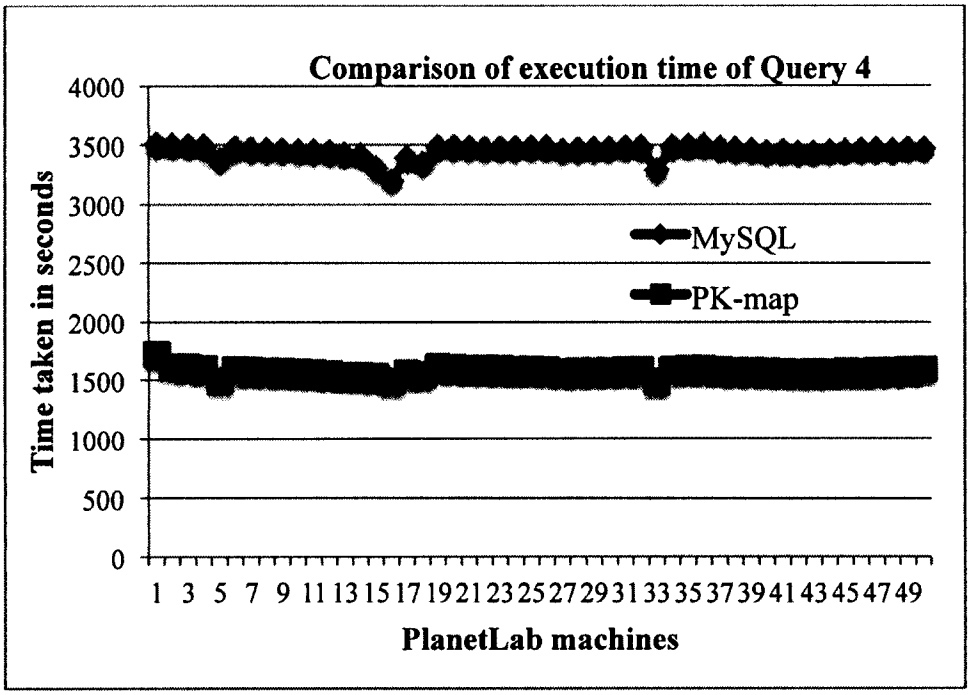


Figure 28: Performance Comparison of Query 4 of Table 6

Figure 27 compares the time taken for processing Query 3 of Table 6. To answer this query, we need to scan the PARTSUPP table, NATION table, NATIONKEY-map, and SUPPKEY-map.

Figure 28 compares the time taken for processing Query 4 of Table 6. This query has an aggregate operation and a filtering constraint on Non-Primary Key of the table ORDERS. Hence, we scan the NATIONKEY-map, CUSTKEY-map, and ORDERS table to process the query.

Figure 29 compares the time taken for processing Query 5 of Table 6. This query is similar to Query 4, but needs to keep track of "count" in order to find the "AVG".

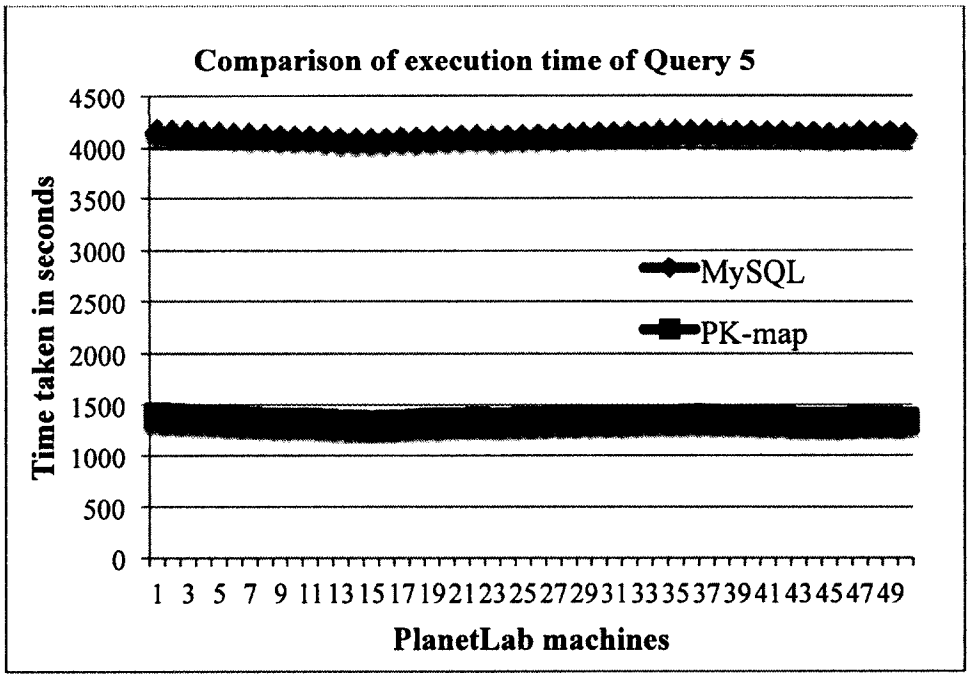


Figure 29: Performance Comparison of Query 5 of Table 6

In the above experiments, MySQL takes 3 inter-machine communications, while our framework takes only one communication. In MySQL, 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 do additional message processing. This will effect the overall time taken by the query. But, with our map structures we do not communicate for every join operation reducing the communication delay. This less interaction between the database machines in the cluster reduces the communication overhead and the effect of poor load balancing. Based on the data in Figure 25 to 29, it is clear that, the query execution time required by the state-of-the-art MySQL is more than our framework.

5. DISCUSSION

In this chapter we will analyze the advantages and disadvantages of some of the techniques used in the proposed framework of Chapter 3. In Section 5.1, we analyze the size of proposed storage structures, size of the inter-node communication, and discuss the tradeoffs between storage space and query performance. In Section 5.2, we look at the advantages of using the broadcast communication method in detail. Finally in Section 5.3, we discuss the data distribution, skew handling and data manipulation.

5.1 ANALYSIS OF STORAGE STRUCTURES

We first see how to find the size of our proposed storage structures and analyze whether they are worth considering compared to improvement in the query performance and decrease in interdependency between the database nodes. Finally we analyze the inter-node communication message size.

5.1.1 PK-MAP AND TUPLE_INDEX-MAP SIZE

As we discussed in the Chapter 3.2, the number of rows of a PK-map is equal to the number of rows of a dimension table (i.e., the primary key table). The 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 data location information (local/remote) might also be stored to locate the table data and direct the data

access request. Thus, the overall size of each map will be,

Size Of PK-map ← $S_1 + \Sigma^n_{i=1} S_2[i] + c$ **(1)**

S_1 (Size of 1st column) ← (Number of rows of PK) * (Size of PK attribute)

S_2 (Size of other columns) ← (Number of rows of PK) * (Size of Record-id)

i.e., S_2 ← (Number of rows of PK) * (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.

'PK' is the Primary Key and 'FK' is the Foregin Key

The number of rows of the Tuple-index-map will be equal to the number of rows of the foreign key table. And the Tuple-index-map will have only one column storing the record-id of the records in the foreign key table. Hence, the size of the Tuple-index-map will be the size of record-id times the number of rows of the foreign key table. Here we can assume that the record-ids can be stored as integers.

Size Of Tuple-index-map ← (Number of rows of FK) * (Size of record-id) **(2)**

For example, consider 5 regions and 25 nations (each region has 5 nations). The PK-maps, REGIONKEY-map and NATIONKEY-map will look like Table 2 and Table 3 of 3.2.1. The Tuple-index-map, REGIONKEY-Tuple-index-map will look like Table 4 of the Chapter 3.2.2. Let us consider the size of the region key be varchar(10) and nation key be varchar(10).

Example:

Table I size: S1 = 50 bytes, S2 = 40 bytes and Size of the RegionKey-map ≈ 90 + c bytes

Table II size: S1 = 250 bytes, S2 = 200 bytes and Size of the NationKey-map ≈ 650 + c bytes

*Table III size: Number of rows of the nation table is 25 and Size of the Tuple-index-map ≈ 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 the 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. The PK-map will grow only when the new primary key is added to the dimension table. Size of the Tuple-index-map will grow proportional to fact table. But, since we only store record-id of foreign key in Tuple-index-map, its size will be manageable.

We analyzed the total size of all the required maps for varying size of data in Figure 15 of the Chapter 4.1.1.1 and found that the total size of map structures will be around 10% to 12% of the actual data size. Thus, each database node in the cluster will need at most 12% of extra storage space. This extra storage space is very small when compared to the advantages these map structures provide. Map structures reduce the time taken for query execution by decreasing the communication overhead and decreasing the inter-

node dependency. They reduce the meta-data maintenance and ease the scale-out / scale-in of database nodes in the cloud.

5.1.2 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 the general distributed query execution, node exchanges 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 Chapter 3.3.1, using our approach nodes communicate only record-ids during the execution of Example 1 query (Figure 8). 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 the region key attribute (varchar). In this case the message size is small, but when results of the large tables need to be exchanged the message size will be significantly larger. Figure 16 of Chapter 4.1.1.2 shows the total size of the intermediate results exchanged during query execution.

5.2 COMMUNICATION METHOD

In a distributed database system, unicast approach is commonly used to exchange data required to perform join/any operation of a query. In a unicast approach data/message is sent to one node in the network with specified address. These distributed

databases store data distribution details in the metadata. Then they use this metadata information to redirect the messages to specific nodes in the cluster.

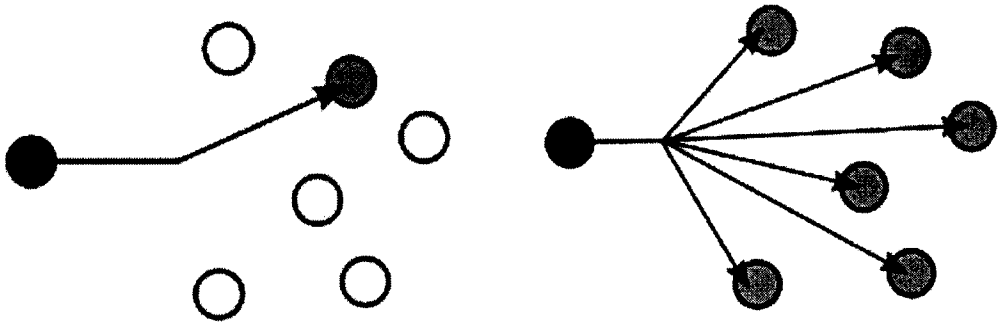


Figure 30: Unicast (left) and Broadcast (right) Communication
(http://en.wikipedia.org/wiki/Routing)

In the Cloud network due to the virtualization and elasticity, it is not feasible to store detailed data distribution information in the meta-data. Instead, it is advantages to use the broadcast method to exchange data/message with all other nodes executing query in parallel. This will be similar to gossip based messaging used by the Cassandra Database system.

In our framework we have considered broadcast type of message exchange. Because, in a broadcast method we need to only store cluster information in the metadata. We do not need detailed data distribution information to do the communication. This decreases the inter-dependency between the nodes executing analytical query in the cluster. Thus, broadcasting approach is not prone to performance degradation in the event of cluster changes, which is very common in cloud architecture.

The disadvantage of the broadcast method is that, it will perform more number of communications than the unicast method. Also, each node will have to filter messages

sent to it. In our approach, we are eliminating most of the communications required for the join operation. We are also reducing the message size of each communication during the query processing. Thus, the above increase in communication using the broadcast messaging will not degrade the performance (as shown in performance study).

5.3 DATA DISTRIBUTION, MANIPULATION AND SKEW HANDLING

In this Section we will analyze the advantages of the proposed storage structures in data distribution, skew handling and data manipulation. We won't analyze our PK-map and Tuple-index-map creation in depth because these maps are created along with the index creation after the first data load operation. Since our maps creation is similar to the index creation, time taken for the creation of map structures is similar to any other index creation.

5.3.1 DATA DISTRIBUTION AND SKEW HANDLING

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 the dataset is called data skew. It has direct impact on the distributed query execution leading to poor load balancing and high response time. As stated in the paper (Walton, 1991), 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 the 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 data skew and poor load balancing.

To overcome the above poor load balancing of the skewed data, commercial databases use different techniques for query processing. One of the techniques is that the query planner will find an estimate about the amount of data that needs to be processed on each node for the given query. Then, they generate a plan to move portion of data from nodes having large amount of data processing to nodes with fewer amounts to data to process. This load balancing can be static or dynamic. In a static architecture, data movement is initiated while generating the query plan as explained above. In a dynamic architecture, data movement is initiated when any of the nodes has finished its local processing. So, the portion of data from heavily loaded node is moved to idle nodes. These architectures improve the performance of the query. But it introduces more communication and bandwidth usage, which will increase the bandwidth and CPU cost of cloud users. So, we need to change the data distribution architecture for Cloud Data Warehouses, which gives similar performance with less communication overhead.

As we discussed earlier, future of databases are to create a Database Management System (DBMS), which is fully virtual and elastic in nature without introducing additional communication overhead. To achieve this, we need to change the architecture

of data storage, data distribution and query processing techniques that has been used from decades for non-virtualized distributed systems. So, in our framework we can consider a random data distribution (such as round robin) with optimizations to storage architecture or metadata to overcome the communication overhead. As shown in the earlier sections, we proposed map structures to keep some information such that we can reduce the communication overhead of query processing. We also designed query-processing algorithms, which will use our map structures and process query efficiently.

Through random data distribution we can reduce the problem of partition skew caused by the join attribute partitioning. Our proposed map structures eliminate the problem caused by random data distribution such as additional data movement and communication overhead (message processing, and bandwidth usage). Since we store the relationship information of the star-schema tables in our maps, we do not need to communicate for every join operation as required by the regular random data distribution architectures. Hence, we reduce the data skew problem, eliminate poor load balancing, and reduce additional communication overhead. In addition, some amount of intrinsic skew is also handled by random distribution.

5.3.2 DATA MANIPULATION

The data manipulation operations like insert, delete and update operations are done along with the batch upload in Cloud Data Warehouses. The data manipulation operations are rare and 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

foreign key logical record-ids, we don't have to update maps when there is an update operation on the data values other than primary key or foreign key attribute. Usually an update operation will occur on non-primary key or non-foreign key attributes. If there is an update operation on primary or foreign key values, then we need to do same operations on map structures as for data tables (delete a row, insert new row, and re-sort if necessary). As our map structures are stored in memory, the memory update operation is always quicker than the disk update operation. Also, in our performance evaluation environment, we store map structures as a binary search tree and hence, each DML operation is in terms of $O(\log_n)$.

For an insert operation on the dimension table, if it is a new record, we can append new row at the end of the PK-map. But, if the record falls in the existing range of primary key values then, we insert it in particular position. Here PK-map might have to be re-sorted.

For an insert operation is on the fact table, we increment the count of corresponding primary key value in the PK-map and add new row to the Tuple-index-map in sorted order. In PK-map, we either can update all the rows after the incremented row, or we can keep the update information in header. By keeping it in header we do not have to manipulate PK-map rows. Hence, when we have to access any row (ex. row 10) we need to check the header whether there is any row below row 10 has updated. If yes, perform arithmetic operation to find the correct start address. Later we can purge all the deleted rows after some interval of time or during the batch update.

For a delete operation on the dimension table, we need to delete a row from the PK-map and corresponding rows from Tuple-index-map (if not deleted already). We also

need to update the PK-map or keep the deletion information in header. We then need to push all the below rows to remove the hole in Tuple-index-map. There is no need of sorting the Tuple-index-map. For a delete operation on the fact table, we decrement the count of corresponding primary key values in the PK-map and update it accordingly. Then, delete corresponding row of the Tuple-index-map, move the rows to fill the hole.

The data manipulation operations are expensive in Data Warehouses. Existing distributed databases pre-process analytical queries to create data cubes and materialized views in order to improve the performance. But, these data cubes and materialized views are not feasible to use these days due to the drastic change in the network architecture and user requirements. Recently, the interval between historical and current data has been reduced. Due to this decrease in the interval, data is more often updated (often batch updates), which makes the re-computation of cubes not feasible. In addition, virtualization of the cloud architecture makes data cube generation and maintenance more difficult.

6. CONCLUSIONS

6.1 SUMMARY

Evolution of distributed systems into cloud services is a major step forward in the computer technology. The cloud services have gained a lot of attention from business and academic communities because of its automatic resource scalability, cost-effective pay-as-you-go model, and no hassle of investment and maintenance. A cloud data warehousing is one such emerging service. Moving from a simple distributed data warehouse system to a cloud data warehouse system introduces some problems, which require change in the database architecture. We have done some analysis and developed techniques that are crucial to make cloud data warehouse dynamically scalable with less interdependency during the query processing.

We analyzed the requirements of cloud services and studied the features of existing distributed shared-nothing data warehouses. We noticed that the shared-nothing architecture of existing data warehouse scales well when incorporated into cloud. But, the data warehouse queries tend to have huge table scans, multi-dimensional joins, and aggregates. These queries need node-to-node communication during the query processing. To reduce this node-to-node communication overhead, data warehouses store node specific data distribution information in their metadata, and they use materialized views and indexes. Storing and using the above information causes problems when incorporated into cloud services. The first problem is that the metadata has to be updated every time

there is a scale-out or scale-in. The second problem is that the number of communication increases every time the number of nodes is increased. Finally, we might have to update the materialized views and global indexes every time the cluster changes.

Considering the above problems, we presented a framework, which improves the performance of distributed analytical queries in cloud data warehouses. Our framework consists of two storage structures, a join query processing algorithm and a query processing algorithm. While designing this framework we considered some of the important bottlenecks during query execution on the cloud such as, metadata maintenance, interdependency between nodes, bandwidth usage for communication, data/message size exchanged and overall time taken by the query.

Proposed storage structures PK-map and tuple-index-map stores relationship information between the tables of a star-schema. In PK-map we store mapping information of the primary key and logical record-id of the Tuple-index map. In Tuple-index-map, we store mapping of logical to actual record-ids of the foreign keys.

We then designed two algorithms for processing analytical queries using the proposed storage structures, reference graph and referential integrity constraints. The first algorithm is used to process the multi-join query. In this algorithm for each join predicate present in the query, we scan PK-map and Tuple-index-maps instead of communicating with the peer nodes and then perform the join operation. The second algorithm is used to process the aggregate query. In this algorithm, we use the join algorithm for evaluating join predicates, we eliminate most of the sort operation, and we perform group-by operation on the fly using our maps.

We analyzed the storage space required by the proposed maps, and data

distribution and skew handling. Then, we presented an extensive performance study on a small-scale network and a large-scale cloud network. For the small-scale network, we used 4 virtual nodes cluster and ran experiments using some of the queries of TPCH benchmark. For the 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 framework improves the performance of the multi-join query in Cloud Data Warehouses and reduces the communication overhead.

6.2 FURUTRE WORKS

In this dissertation, we have proved that with our proposed framework we can solve some of the problems of existing distributed data warehouses and can incorporate it into cloud services. We have used the TPC-H benchmark data generator to generate data to conduct experiments. However, it is also important to run some experiments using real world application data. TPC-H generates data using the Scale Factor (SF) and the ratio between primary key to foreign key records is evenly distributed [70]. This might not be the case in real world applications. The fact table might not have records for some primary keys of the dimension table, or they might have highly uneven number of foreign key records for some primary keys. We think that, these uneven data distribution in the real world data will have higher performance benefits using our framework. Hence, we are working on running experiments using real world data such as, movie release and sales data, historical stock market data, market basket data, etc.

Other areas we are thinking to analyze are, incorporating our framework into a commercial data warehouse product, analyzing metadata updating during the scale-out

and scale-in of resources, and analyzing the batch update process.

7. SUPPORTING PUBLICATIONS

This thesis work has been published in the following peer reviewed journal and conferences:

1) Swathi K., Tingjian Ge., Benyuan L., and Cindy X. C. **Communication Cost Optimization for Cloud Data Warehouse Queries.** In Proceeding, 4th IEEE CloudCom, pages 512-519, Taipei, Taiwan, 2012
2) Swathi K., Tingjian Ge., Xinwen F., Benyuan L., and Cindy X. C. **Optimizing Communication for Multi-Join Query Processing in Cloud Data Warehouses.** In Journal, International Journal of Grid and High Performance Computing (IJGHPC), volume 5, Issue 4, 2013
3) Swathi K., Tingjian Ge., Xinwen F., Benyuan L., Amrith K., and Cindy X. C. **Optimizing Aggregate Query Processing in Cloud Data Warehouses.** In Proceeding, DEXA's 7th International Conference on Data Management in Cloud, Grid and P2P Systems (Globe). Munich, Germany 2014

8. LITERATURE CITED

[1] AmazonEC2: http://aws.amazon.com/ec2/

[2] Google Cloud: http://www.google.com/enterprise/cloud/storage/

[3] EMC Greenplum: http://www.greenplum.com/products/greenplumdatabase

[4] IBM: http://www-01.ibm.com/software/data/infosphere/warehouse/

[5] Microsoft: http://msdn.microsoft.com/enus/library/aa197903(v=sql.80).aspx

[6] Microstrategy: http://www.microstrategy.com/

[7] OracleBI: http://www.oracle.com/us/solutions/entperformancebi/enterprise-edition-066546.html

[8] SqlServer: http://technet.microsoft.com/enus/sqlserver/cc510300.aspx

[9] TPC-H: http://www.tpc.org/tpch/spec/tpch2.14.4.pdf.

[10] Vertica: http://184.106.12.19/wpcontent/uploads/2011/01/VerticaArchitectureWhitePaper.pdf

[11] D. J. Abadi. Data management in the cloud: Limitations and opportunities. IEEE Data Eng. Bulletin, 2009, volume 32 (1), pages 3-12.

[12] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column Oriented Database Systems. In Proceeding, VLDB 2009, volume 2 (2), pages 1664-1665.

[13] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In Proceeding, ACM SIGMOD 2008, pages 967-980, New York, NY, USA.

[14] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. In Proceeding, VLDB 2009, volume 2 (1), pages 922-933.

[15] M. O. Akinde, M. H. Bohlen,T. Johnson, L. V. Lakshmanan, and D. Srivastava. Efficient olap query processing in distributed data warehouses. In Proceeding, EDBT 2002, pages 336-353.

[16] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. In Proceeding, VLDB 2009, volume 2 (1), pages 277-288.

[17] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. In Proceeding, VLDB 2008, volume 1(1), pages 502-513.

[18] M. Ivanova, M. L. Kersten, and N. Nes. Self-organizing strategies for a column-store database. In Proceeding, EDBT 2008, pages 157-168, New York, NY, USA.

[19] R. MacNicol and B. French. Sybase iq multiplex designed for analytics. In Proceeding, VLDB 2004, pages 1227-1230.

[20] D. Papadias, and P. Kalnis. Multi-query optimization for on-line analytical processing. In Journal, Information Systems 2003, Volume 28 (5), pages 457-473.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column oriented dbms. In Proceeding, VLDB 2005, pages 553-564.

[22] H. Wang, X. Meng, and Y. Chai. Efficient data distribution strategy for join query processing in the cloud. In Proceeding, CloudDB 2011, pages 15-22, New York, NY, USA.

[23] W. Hasan, and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In Proceeding, VLDB 1994, Santiago, Chile.

[24] J. Li, A. Deshpande, and S. Khuller. Minimizing Communication Cost in Distributed Multi-query Processing. In Proceeding, ICDE 2009, Shanghai, China.

[25] A. Kemper, and T. Neumann. Hyper: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In Proceeding, ICDE 2011, pages 195-206, Hannover, Germany

[26] A. Vlachou, C. Doulkeridis, K. Norvag, and Y. Kotidis. Peer-to-Peer Query Processing over Multidimensional Data. Springer 2012.

[27] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the cloud. In Proceeding, CIDR 2011, pages 235-240, California, USA.

[28] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In Proceeding, ACM SIGMOD 1996, volume 25, pages 57-67, Montreal, Canada.

[29] D. Kossmann. The state of the art in distributed query processing. In Journal, ACM CSUR 2000, volume 32 (4), pages 422-469, New York, NY, USA.

[30] D. Xin, J. Han, X. Li, and B. W. Wah. Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration. In Proceeding, VLDB 2003, volume 29, pages 476-487, Berlin, Germany.

[31] D. Boukhelef, and H. Kitagawa. Efficient Management of Multidimensional Data in Structured Peer-to-Peer Overlays. In Proceeding, VLDB 2009, volume 35, Lyon, France.

[32] G. Graefe. New algorithms for join and grouping operations. In Journal, Computer Science - Research and Development 2012, volume 27 (1), page 3-27.

[33] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In Proceeding, USENIX FAST 2009, pages 71-84, San Francisco, California, USA.

[34] H. G. Molina, and K. Salem. Main Memory Database Systems: An Overview. In Journal, IEEE TKDE 1992, volume 4 (6), pages 509-516.

[35] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In Proceeding, ACM SIGMOD 2009, pages 1-2, Providence, USA.

[36] L. Peterson, and T. Roscoe. The design principles of PlanetLab. In Journal, ACM SIGOPS 2006, volume 40 (1), pages 11-16, New York, USA.

[37] Planetlab Cloud, http://www.planet-lab.org/

[38] S. Chaudhuri, and K. Shim. Including Group-By in Query Optimization. In Proceeding, VLDB 1994, pages 354-366, Santiago, Chile.

[39] Survey, http://www.oracle.com/us/products/database/high-performance-data-warehousing-1869944.pdf

[40] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimentional Aggregates. In Proceeding, VLDB 1996, volume 22, pages 506-521, Mumbai (Bombay), India.

[41] S. Kurunji, T. Ge, B. Liu, and C. X. Chen. Communication Cost Optimization for Cloud Data Warehouse Queries. In Proceeding, IEEE CloudCom 2012, pages 512-519, Taipei, Taiwan.

[42] S. Bellamkonda, H. Li, U. Jagtap, Y. Zhu, V. Liang, and T. Cruanes. Adaptive and Big Data Scale Parallel Execution in Oracle. In Journal, VLDB 2013, volume 6 (11), pages 1102-1113.

[43] T. Neumann, and G. Moerkotte. A Combined Framework for Grouping and Order Optimization. In Proceeding, VLDB 2004, volume 30, pages 960-971, Toronto, Canada.

[44] Teradata, http://www.teradata.com/white-papers/Teradata-Aggregate-Designer-eb6110

[45] W. P. Yan, and P. Larson. Eager Aggregation and Lazy Aggregation. In Proceeding, VLDB 1995, pages 345-357, Zurich, Switzerland.

[46] W. Hasan, and R. Motwani. Coloring Away Communication in Parallel Query Optimization. In Proceeding, VLDB 1995, pages 239-250, Zurich, Switzerland.

[47] X. Wang, and M. Cherniack. Avoiding Sorting and Grouping in Processing Queries. In, Proceeding, VLDB 2003, volume 29, pages 826-837, Berlin, Germany.

[48] Y. Cao, R. Bramandia, C. Y. Chan, and K. L. Tan. Sort-Sharing-Aware Query Processing. In Journal, VLDB 2012, volume 21 (3), pages 411-436.

[49] Paraccel: http://h21007.www2.hp.com/portal/download/partner/Corporate%20Brief%20on%20ParAccel%20Data%20Warehousing%20v6_1236198035276.pdf

[50] Y. Chen, R. L. Cole, W. J. McKenna, S. Perfilov, A. Sinha, and E. S. Jr. Partial join order optimization in the paraccel analytic database. In Proceeding, ACM SIGMOD 2009, pages 905–908.

[51] SAP HANA: http://www.saphana.com/welcome

[52] Y. Cao, R. Bramandia, C. Y. Chan, and K. L. Tan. Optimized Query Evaluation Using Cooperative Sorts. In Proceeding, ICDE 2010, pages 601-612, Long Beach, CA, USA.

[53] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In Proceeding, ACM SIGMOD 2011, pages 961-972, Athens, Greece.

[54] Arshdeep Bahga, Vijay Madisetti. Cloud Computing: A Hands-On Approach.

[55] Amazon S3 Service Level Agreement: http://aws.amazon.com/s3-sla/

[56] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. IEEE Communications

Surveys & Tutorials, 2011, volume13 (3), Pages 311 – 336

[57] Yingjie Shi, Xiaofeng Meng, Jing Zhao, Xiangmei Hu, Bingbing Liu, Haiping Wang. Benchmarking cloud-based data management systems. In Proceedings, Second international workshop on Cloud data management (CloudDB), 2010, pages 47-54

[58] Islam M. A., and Vrbsky S. V. Tree-Based Consistency Approach for Cloud Databases. IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), November 2010, Indianapolis, USA, pages 401 – 404

[59] Patrick Valduriez. Principles of Distributed Data Management in 2020? Database and Expert Systems Applications (DEXA) August 2011, Toulouse, France, pages 1-11

[60] Peter Mell, and Timothy Grance. The NIST definition of Cloud Computing. 16th and final version of National Institute of Standards and Technology (NIST) Special Publication 800-145, September 2011

[61] Gartner 2015 tred list: http://www.tnooz.com/article/gartner-updates-top-strategic-technology-trends-2015/

[62] Gartner: http://www.gartner.com/technology/research/top-10-technology-trends/

[63] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Communications of the ACM Magazine, April 2010, volume 53 (4), pages 50-58.

[64] Google Cloud Database Services: https://cloud.google.com/appengine/docs/python/storage#app_engine_datastore

[65] Amazon Cloud Database Services: http://aws.amazon.com/products/

[66] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM Magazine, Special 25th Anniversary Issue, January 1983, volume 26 (1), pages 64-69

[67] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. International Conference on Data Engineering, 2008.

[68] S. Madden, D. DeWitt, and M. Stonebraker. Database parallelism choices greatly impact scalability. DatabaseColumn Blog. www.databasecolumn.com/2007/10/database-parallelism-choices.html

[69] D. DeWitt and M. Stonebraker. MapReduce: A major step backwards. Database Column Blog: www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html

[70] TPC-H data generator analysis: http://kejser.org/tpc-h-schema-and-indexes/

[71] TPC-DS Benchmark: http://www.tpc.org/tpcds/spec/tpcds_1.2.0.pdf

[72] Joao Leitao, Jose Pereira, Luis Rodrigues. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. IEEE/IFIP 37th International Conference on Dependable Systems and Networks (DSN), June 2007, Edinburgh, UK, pages 419-429

[73] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In Proceeding, VLDB 2008, Auckland, New Zealand