IMPROVING THE PERFORMANCE OF HADOOP/HIVE BY SHARING SCAN AND COMPUTATION TASKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

SERKAN ÖZAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2013

Approval of the thesis:

IMPROVING THE PERFORMANCE OF HADOOP/HIVE BY SHARING SCAN AND COMPUTATION TASKS

submitted by **SERKAN ÖZAL** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Ahmet Coşar
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Ahmet Coşar
Computer Engineering Dept., METU _____

Prof. Dr. Adnan Yazıcı
Computer Engineering Dept., METU _____

Prof. Dr. Özgür Ulusoy
Computer Engineering Dept., Bilkent University _____

Asst. Prof. Dr. İsmail Sengör Altıngövde
Computer Engineering Dept., METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SERKAN ÖZAL

Signature :

ABSTRACT

IMPROVING THE PERFORMANCE OF HADOOP/HIVE BY SHARING SCAN AND COMPUTATION TASKS

Özal, Serkan

M.S., Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Ahmet Coşar

June 2013, 53 pages

MapReduce is a popular model of executing time-consuming analytical queries as a batch of tasks on large scale data. During simultaneous execution of multiple queries, many opportunities can arise for sharing scan and/or computation tasks. Executing common tasks only once can reduce the total execution time of all queries remarkably. Therefore, we propose to use Multiple Query Optimization (MQO) techniques to improve the overall performance of Hadoop Hive, an open source SQL-based distributed warehouse system based on MapReduce. Our framework, SharedHive, transforms a set of correlated HiveQL queries into new global queries that can produce the same results in remarkably smaller total execution times. It is experimentally shown that SharedHive outperforms the conventional Hive by %20-90 reduction, depending on the number of queries and percentage of shared tasks, in the total execution time of correlated TPC-H queries.

Keywords: Hadoop, Hive, Multiple-query Optimization, Distributed Data Warehouse

ÖZ

TARAMA VE HESAPLAMA İŞLERİNİN PAYLAŞTIRILMASIYLA HADOOP/HIVE ÜZERİNDE PERFORMANS İYİLEŞTİRİMİ

Özal, Serkan

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Assoc. Prof. Dr. Ahmet Coşar

Haziran 2013 , 53 sayfa

MapReduce zaman alan analitik sorguların büyük ölçekli veriler üzerinde toplu olarak işletilmesi için popüler bir yöntemdir. Çoklu sorguların eşzamanlı işletilmelerinde, tarama ve hesaplama işlemleri için birçok yöntem kullanılabilmektedir. Ortak kısımların sadece bir kere işletilmesi toplam işlem süresini önemli ölçüde düşürebilmektedir. Buradan yola çıkarak, biz Hadoop üstünde çalışan açık kaynak kodlu SQL tabanlı dağıtık veri ambarı yönetim sistemi olan Hive framework'ü ile çalışan Çoklu Sorgu İyileştirimi (ÇSİ) yöntemi öneriyoruz. Bizim framework'ümüz, SharedHive, benzer kısımları bulunan HiveQL sorgularını ortak kısımların birlikte kullanılmasını sağlayarak genel HiveQL sorgularına dönüştürmektedir. Bu sayede toplam işletim süresinde önemli iyileşmeler görülebilmektedir. SharedHive ile ortak kısımları bulunan TPC-H sorgularında toplam işletim süresi olarak %20-90 arasında iyileştirme sağlanabilmektedir.

Anahtar Kelimeler: Hadoop, Hive, Çoklu-Sorgu İyileştirimi, Dağıtık Veri Ambarı

Dedicated to all my family

ACKNOWLEDGMENTS

I would like to thank my supervisor Associate Professor Ahmet Coşar for his constant support, guidance and friendship. It was a great honor to work with him and our cooperation influenced my academical and world view highly. I also would like to thank Tansel Dökeroğlu for his support and guidance. He also motivated and influenced me highly in scientific context.

My family also provided invaluable support for this work. I would like to thank specially to my parents, Periha Özal and Ahmet Özal, and my dear sister Yeşim Özal. In addition, I am so grateful for all the love and support given by my aunt Yurdagül Özal and my grandfather Sadık Özal throughout my life.

And finally very special thanks to a group of people who've taught me real meaning of friendship lately. Special thanks to my colleagues, İbrahim Gürses, Barış Cem Şal, Mert Çalışkan and Yusuf Erdem.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

SQL	Structured Query Language
MQO	Multiple Query Optimization
HOD	Hadoop on Demand
HDFS	Hadoop Distributed File System
HiveQL	Hive Query Language
CLI	Command Line Interface

CHAPTER 1

INTRODUCTION

1.1 Large Scale Data Processing

Processing large-scale data in the amounts of hundreds of terabytes is a very difficult task. Solving the problems associated with high volume data requires can be achieved by dividing the data and work to many computers that will all work together in parallel to complete the task in a reasonable time. However, the use of so many computers for co-operatively solving this problem increases the risk of any one of these machines failing before the data processing task completed, ruining the work done by the other computers as well. The probability of any machine failing in a given time interval is small but when hundreds of computers are involved the probability because quite high. In such a environment the system designers must expect any large task to have at least one failure before completing and take precautions to handle this case. Another kind of failures is network failures where a network device such as a switch or router.Network failures may cause data to be lost or arrive with arbitrary delays. Hard disk failures, overheating problems and memory CRC errors are also possible. Software library and network protocol mismatch errors may occur over long executing tasks and operating systems. Memory leaks, buggy softwares not releasing shared resource locks (causing deadlocks) and not properly closing files may also cause co-operation between local and remote processes to fail. No matter what combination of such errors occur the failing node must not prevent the remaining machines to continue doing useful work and eventually successfully completing the parallel big data tasks. Over the decades many operating systems have been developed to address each of these different kinds of failures but usually have ignored most of the failure types other than the one they were designed to solve. The Hadoop system does not provide protection against any specific failure and cannot guarantee correct operation if the software itself is buggy or has been compromised. Hadoop will detect a computing node that has been unresponsive due to some failure and resolve the problem and continue the computation of big data task. A big data computation will typically have inputs of several terabytes which could be stored on a single machine's hard disk. However, intermediate operations could multiply the sizes of intermediately generated data and this could easily exceed the capacity of a single machine requiring automatic distribution onto the available computing machines. The distribution such a large amounts of data requires very large network bandwidths which could be available on a gigabit LAN but distant cluster machines on separate racks would not have such a high space connection between them.

A distributed system would typically use RPC requests for coordinating execution of big data tasks on many machines. However, unless carefully controlled unrestricted execution of such RPC requests between many pairs of compute nodes could easily saturate the network and

introduce unacceptable delays. In a big data compute system where 100 compute nodes are collaborating a big data processing task, if one of the nodes fails the remaining 99 nodes must be able to still complete their parts of the task and also allocate one of them to re-compute the task that was originally assigned to the failing node. Thus, the computations on 99 nodes are not lost and a minimal penalty of recomputing 1/100 of the big data task with minimal negative affect on the whole computation.

1.2 The Motivation of the Study

When users want to benefit from both MapReduce and SQL interface, mapping SQL statements to MapReduce tasks can become a very difficult job [9]. Hive does this work by translating queries to MapReduce jobs, thereby exploiting the scalability of Hadoop while presenting a familiar SQL abstraction [10]. These attributes of Hive make it a suitable tool for data warehouse applications where large scale data is analyzed, fast response times are not required, and data is not updated frequently [4].

Because most data warehouse applications are implemented using SQL based RDBMSs, Hive lowers the barrier for moving these applications to Hadoop, thus, people who already know SQL can use Hive easily. Similarly, Hive makes it easier for developers to port SQL-based applications to Hadoop. Since Hive is based on *query-at-a-time* model and processes each query independently, issuing multiple queries in close time interval decreases performance of Hive due to its execution model. From this perspective, it is important that there has not been any study that incorporates the Multiple-query optimization (MQO) technique [11, 12, 13] for Hive to reduce the total execution time of the queries.

Studies concerning MQO on traditional warehouses have shown that it is an efficient technique that increases the performance of time-consuming decision support queries remarkably [2, 14, 15, 16]. In order to improve the performance of Hadoop Hive in massively issued query environments, we propose SharedHive which processes HiveQL queries as a batch and improves the total execution time by merging correlated queries before passing them to Hive query optimizer [10, 17, 18]. Our contributions in this study can be listed as:

1. MQO technique is used by an SQL-to-MapReduce translator for the first time.
2. The execution plans of correlated HiveQL queries are analyzed, merged (with an optimization algorithm), and executed together.
3. The developed model is introduced as a novel component for Hadoop Hive architecture.

In Chapter 2, we give information related work on MQO, other SQL-to-MapReduce translators that are similar to Hive, and recent query optimization studies on MapReduce framework. Chapter 3 explains traditional architecture of Hive and introduces our novel MQO component. Chapter 4 explains the process/algorithm of generating a global plan from correlated queries. Chapter 5 discusses the experiments conducted for evaluating SharedHive framework. Finally our concluding remarks are given in Chapter 6.

CHAPTER 2

RELATED WORK

2.1 The Hadoop Approach

HADOOP [1] is a popular open source software framework that allows distributed processing of large scale data sets. It employs MapReduce paradigm [2] to divide the computation tasks into parts (Figure 2.1) that can be distributed to a cluster of computers therefore, providing horizontal scalability [3, 4, 5, 6].

Current technology can build powerful machines with very large data processing capacities by employing cheap low price computers that can all work in parallel. A large single super-computer with 1000 CPUs would be prohibitively expensive compared to a cluster machine containing 1000 single core or 250 quad-core machines to obtain the same processing power. The Hadoop system architecture enables all of these cheap computers to work in collaboration on very large amounts of data and form a very powerful data processing system. In order to allow parallel processing of large data on all CPUs, data are distributed uniformly to all computers while it is being loaded onto the Hadoop system. The Hadoop Distributed File System (HDFS) automatically splits big data files into fragments each having a user-defined size and can be processed independently on a dedicated machine. In order to enable recovery from a failing node without loss of data each fragment of a data file is also replicated (the default is three replicas). The Hadoop system continuously monitors the nodes and when it discovers that one of them has failed the replicas will be re-arranged to reach the same number of replicas again and are still accessible by all nodes. In order to have maximum parallelism HDFS data are record-oriented and each Hadoop operation creates new records from the input sets of data records. Since the amount of data is much and the computation tasks are very simple map and reduce operations it is very easy to move a computation to any node and restart a failed operation on one of the available replicas when a node fails. In order to eliminate unnecessary network traffic the data to be processed must be preferred to be stored on the hard disk of the local machine.

2.2 Map-Reduce

The Map-Reduce framework is introduced by Google.

> A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this

interface that achieves high performance on large clusters of commodity PCs (definition by Google paper on Map-Reduce).

The Map-Reduce framework broadly consists of two mandatory functions to implement: "Map and reduce" (Figure 2.1). A *map* is a function which is executed on each key-value pair from an input split, does some processing and emits again a key and value pair. After map is completed and before *reduce* can begin there is a phase called *shuffle* which copies and sorts the output on key and aggregates values of matching key values using a function (e.g. count, sum, avg, etc.).

These key and "aggregated value" pairs are captured by reduce and outputs a reduced key value pair. This process is also called aggregation as you get values aggregated for a particular key as input to reduce method. Again in reduce you may play around as you want with key-values and what you emit now is also key value pairs which are dumped directly to a file.

Now simply by expressing a problem in terms of map-reduce we can execute a task in parallel and distribute it across a broad cluster and be relieved of taking care of all complexities of distributed computing. Indeed "Life made easy", had you tried doing the same thing with MPI libraries you can appreciate the complexity there scaling to thousands or even hundreds of nodes.

There is a lot more happening in map-reduce than just map-reduce. But the beauty of the hadoop is that it takes care of most of those things and a user need not dig into details for simply running a job, though it would be useful if one has knowledge of those features and can help in tuning parameters and thus improve efficiency and fault tolerance.

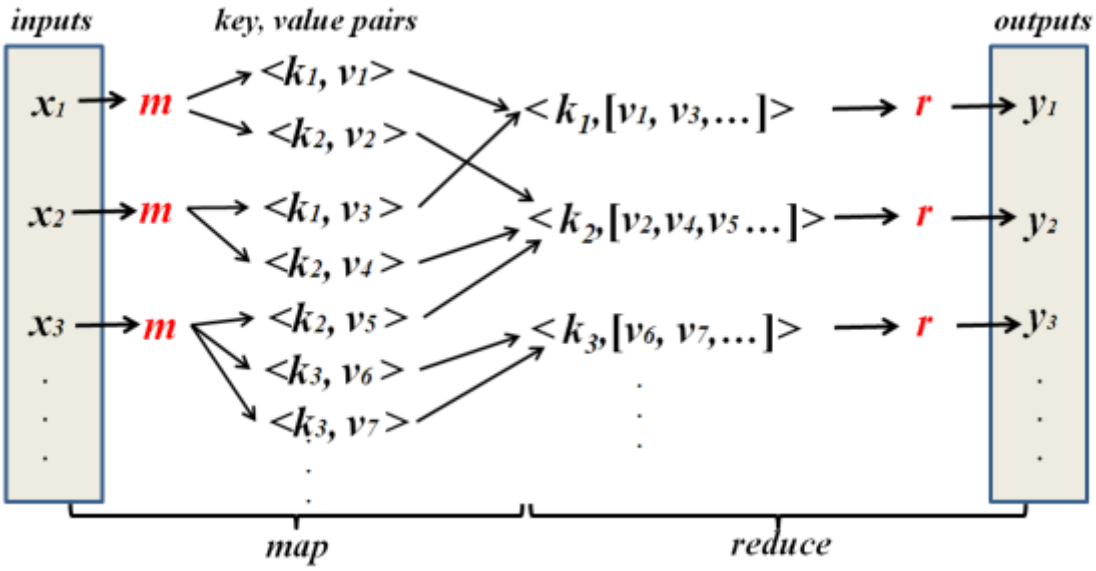

Figure 2.1: MapReduce tasks. x values are input splits and y values are output splits

2.3 Map-Reduce in Hadoop

The performance success of Hadoop is achieved by its dividing input data to sets of records and performing very little communication between nodes because each node performs its operation in isolation from other nodes. The simplicity of these operations what makes the Hadoop framework so powerful. Hadoop uses a programming model called "MapReduce" where the input records are first processed by Mapper tasks (Figure

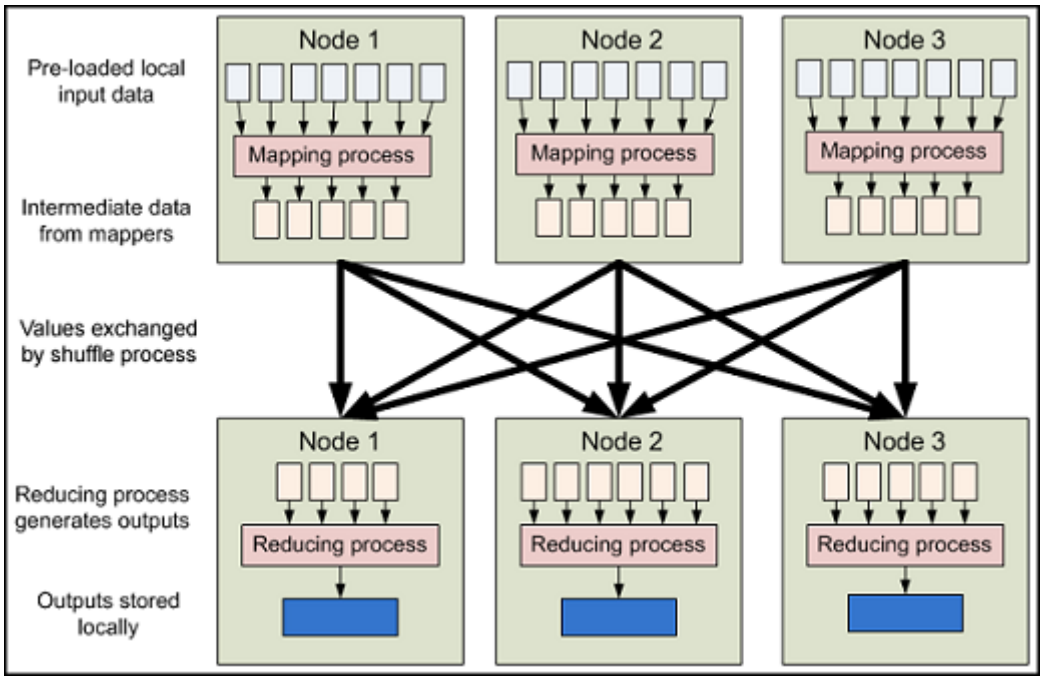

Figure 2.2: Map and reduce tasks run on nodes where individual records of data.

From Yahoo Map-Reduce Tutorial ([52])

Unlike traditional data processing systems where communication occurs explicitly by using MPI calls and data streams between nodes, Hadoop uses key-value pairs to decide the destination nodes of the output of each operation. The Hadoop system manages allocation of data and Map-Reduce operations on computation nodes. When a node fails its task can assigned to a new node and restarted with the same inputs (which are also replicated so are not lost). There is no need for complex message exchanges and recovery/checkpoint operations.

2.4 Hadoop Distributed File System (HDFS)

Hadoop has its own implementation of distributed file system called Hadoop Distributed File System, which is coherent and provides all facilities of a file system. It implements ACLs and provides a subset of usual UNIX commands for accessing or querying the filesystem and if one mounts it as a *fuse dfs* then it is possible to access it as any other Linux filesystem with standard unix commands.

Hadoop Distributed File System (HDFS) is the underlying file system of Hadoop MapReduce. Because of its simplicity, scalability, fault-tolerance, and efficiency Hadoop has gained significant support from both industry and academia. However, it has some limitations on its interfaces and performance [7]. Querying the data with Hadoop as if in a traditional RDBMS infrastructure is one of the most common problems that Hadoop users face. This affects a majority of users who are not familiar with internal details of Map Reduce jobs to extract information from their data warehouses.

2.5 Namenode

A namenode stores information about HDFS file system and it forms a single point of failure for an HDFS installation. It contains information regarding a block's location as well as the information of entire directory structure and files. By saying it is a single point of failure - it is meant that, if namenode goes down - whole filesystem will be unreachable. Hadoop also has a secondary namenode which contains edit log, which in case of a failure of namenode can be used to replay all the actions of the filesystem and thus restore the state of the filesystem. A secondary namenode regularly contacts namenode and takes checkpointed snapshot images. At any time of failure these checkpointed images can be used to restore the namenode. Current efforts are going on to have high availability for Namenode.

2.6 Datanode

A Datanode on HDFS stores actual blocks of data and stores and retrieves them when asked. They periodically report back to Namenode with a list of blocks they are storing.

2.7 Jobtracker and Tasktracker

There is one JobTracker(is also a single point of failure) running on a *master node* and several TaskTrackers running on slave nodes. Each TaskTracker has multiple task-instances running and every task tracker reports to JobTracker in the form of a heartbeat at regular intervals which also carries message of the progress of the current job it is executing and idle if it has finished executing. JobTracker schedules jobs and takes care of failed ones by re-executing them on some other nodes.

2.8 Hive

Hive, an open source SQL-based distributed warehouse system is proposed to solve problems mentioned above by providing SQL like abstraction on top of Hadoop framework (Figure 2.3). Hive is a SQL-to-MapReduce translator and has an SQL dialect, HiveQL, for querying data stored in a cluster [41]. In terms of storage Hive can use any file system supported by Hadoop, although HDFS is by far the most common.

Hive provides its own query language HiveQL (similar to SQL) for querying data on a Hadoop cluster. It can manage data in HDFS and run jobs in MapReduce without translating the

queries into Java. When MapReduce jobs are required, Hive doesn't generate Java MapReduce programs. Instead, it uses built-in, generic Mapper and Reducer modules that are driven by an XML file representing the "job plan". In other words, these generic modules function like mini language interpreters and the "language" to drive the computation is encoded in XML. Hive Queries are translated to a graph of Hadoop MapReduce jobs that get executed on your Hadoop grid. Hive Query Language (HQL) is based on SQL, and there are many of the familiar constructs such as "SHOW", "DESCRIBE", "SELECT", "USE" and "JOIN". Similar to an RDBMS in Hive there are "Databases" that contain one or more "Tables" that contain some data defined by a "Schema".

Hive also supports User Defined Functions (UDFs) and Serialization/Deserialization functions (SerDe's). UDFs allow programmers to write functions to abstract common tasks in Hive. It also allows you to seamlessly connect a Query Language in Hive with functional, procedural or scripting languages. SerDe's allow for the management of arbitrarily structured or unstructured data in Hive. One particularly useful and popular SerDe is the now built-in one for Avro.

Hive uses a metadata store to keep the data warehouse information that links Hive and the raw data. Data loaded into Hive can be maintained in an external location or a copy made into the Hive warehouse. In this way you can create a self-contained data warehouse, or share the infrastructure with MapReduce and Pig programmer.

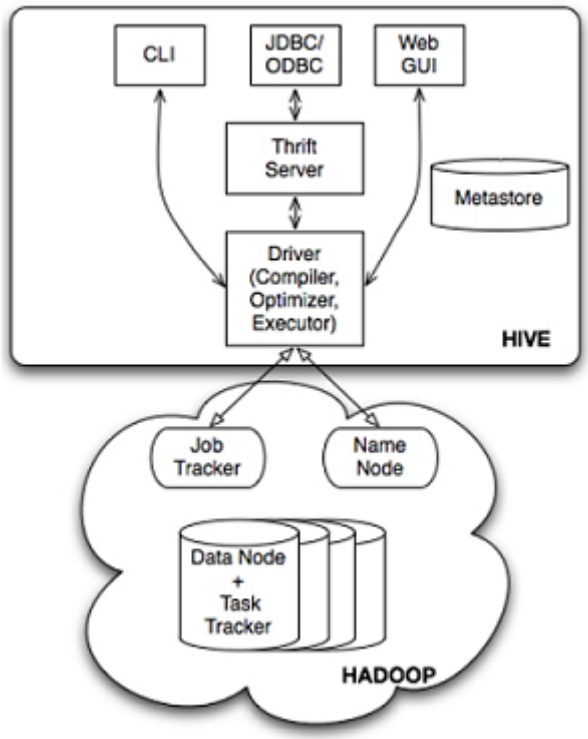

Figure 2.3: Hive Architecture.

Hive can be accessed via a command line and Web User interfaces. Hive also can be used through the JDBC or ODBC APIs provided. The Thrift server exposes an API to execute

HiveQL statements with a different set of languages (PHP, Perl, Pyhton and Java).

The Metastore component is a system catalogue that contains metadata regarding tables, partitions and databases.

It is in the Driver and in the Compiler components that most of the core operations are made. They parse, optimize and execute queries. The SQL statements are converted to a graph (a DAG graph actually) of map/reduce jobs in run time, and this are run in the Hadoop cluster.

2.9 Multiple Query Optimization on Cloud

Hadoop emerged as a cost-effective way of dealing with large scale data [46, 47, 48]. Hadoop is implemented with Java programming language and an Apache top-level project being built and used by a global community of contributors. Hadoop implements a computational paradigm called MapReduce, a particular programming model, for decomposing the computation tasks into junks that can be assigned to a cluster of commodity. Therefore Hadoop provides an efficient and scalable system. Hadoop Distributed Filesystem (HDFS) is the underlying file system. The filesystem is pluggable and there are many other available open source distributed filesystems [49]. However, when the infrastructure is based on conventional RDMBS and the Structured Query Language (SQL) a challenging problem emerges. How will the Hadoop tackle with these queries issued as SQL statements? How will the large number of user that are familiar with SQL use SQL to extract information from their data warehouses?

Hive [50], an efficient SQL-to-mapreduce converter, is the solution of this problem. Hive is a data warehouse system for Hadoop facilitating data analysis, ad-hoc queries, and the analysis of large scale data stored in HDFS. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. At the same time this language also allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express this logic in HiveQL. SQL is widespread, effective, reasonably an intuitive model for organizing and using data. Mapping these familiar data operations to the low-level MapReduce Java API can be boring. Hive does this work and translates most queries to MapReduce jobs, thereby exploiting the scalability of Hadoop, while presenting a familiar SQL abstraction.

Multiple query optimization (MQO) problem was introduced in 1980s and finding an optimal global query plan by using MQO was shown to be an NP-Hard problem [11, 19]. Since then, considerable amount of work was done on RDBMSs and data analysis applications [20, 21]. The problem of identifying common subexpressions is an NP-hard problem [51]. Therefore, Jarke indicates that multirelation subexpressions can only be addressed heuristically [51]. Finkelstein shows how an ad hoc query may be improved by comparing an incoming query with materialized results (intermediate results and final answer) produced from earlier queries [19]. He deals only with equivalent expressions. Jarke discusses the common subexpression isolation in relational algebra, domain relational calculus, and tuple relational calculus. Chakravarthy and Minker identify the equivalence and subsumption of two expressions at the logical level, using heuristics [22]. Rosenthal and Chakravarthy use an and/or graph to represent queries and detects subsumption by comparing each pair of operator nodes from distinct queries [11]. Another issue in MQP is the representation and the processing of multiple queries. The multigraph is proposed for representing multiple Select-Project-Join type queries in [22]. This multigraph can facilitate query processing by using Ingres' instantiation and substitution [22].

In [23], the multigraph was modified for representing the initial state of multiple queries.

Mehta and DeWitt considered CPU utilization, memory usage, and I/O load variables in a study during planning multiple queries to determine the degree of intra-operator parallelism in parallel databases to minimize the total execution time of declustered join methods [22]. A proxy-based infrastructure for handling data intensive applications is proposed by Beynon [23]. This infrastructure was not as scalable as a collection of distributed cache servers available at multiple back-ends. Chen et al. considered the network layer of a data integration system and reduced the communication costs by a multiple query reconstruction algorithm [24]. IGNITE [25] and QPipe [26] are important studies that use the micro machine concept for query operators to reduce the total execution time of a set of queries. A novel MQO framework is proposed for the existing SPARQL query engines in [28]. Yasin et al. designed a cascade-style optimizer for Scope, Microsoft's system for massive data analysis [28]. In recent years, a significant amount of research and commercial activity has focused on integrating MapReduce and structured databases technologies. Mainly there are two approaches: Either adding MapReduce features to parallel database or adding databases technology to MapReduce. The second approach is more attractive because there exists no widely available open source parallel database system whereas MapReduce is available as an open source project. Furthermore, MapReduce is accompanied by a plethora of free tools as well as cluster availability and support. Hive [41], Pig [29], Scope [15], and HadoopDB [7, 30] are the projects that provide SQL abstractions (SQL-to-MapReduce translators) on top of MapReduce platform to familiarize the programmers with complex queries. SQL/MapReduce [31] and Greenplum [16] are recent projects that use MapReduce to process user-defined functions (UDF). Recently, there are interesting studies to apply MQO to MapReduce frameworks for unstructured data. MRShare [32] is one of these studies that processes a batch of input queries as a single query. The optimal grouping of queries for execution is defined as an optimization problem based on MapReduce cost model. The experimental results reported for MRShare demonstrate its effectiveness. In spite of some initial MQO studies to reduce the execution time of MapReduce-based single queries [33], to the best of our knowledge there is no study like ours that is related to optimize the execution time of multi-queries on SQL-to-MapReduce translator tools.

CHAPTER 3

SHAREDHIVE SYSTEM ARCHITECTURE

In this Chapter, we give brief information about architecture of SharedHive which is the modified version of Hadoop Hive with new MQO component as shown in Figure 3.1. Inputs to compiler-optimizer-executer are pre-processed by a Multiple Query Optimizer component which examines incoming queries and produces a single HiveQL command to execute a group of correlated queries. System catalog and relational database structure (relations, attributes, partitions, etc.) are stored and maintained by *Metastore*. Once a HiveQL statement is submitted, it is maintained by *Driver* which controls the execution of tasks to answer the query. First, a directed acyclic graph is produced by HiveQL to define the MapReduce tasks to be executed. Next, the tasks are executed.

3.1 Query Processing for Multiple Query Optimization

HiveQL statements are submitted via the Command Line Interface (CLI) or the Web User Interface ([42]). Normally the query is directed to the driver component in a conventional Hive architecture. In the architecture we proposed, MQO component receives the incoming queries before the driver component. The set of the incoming queries are inspected, their common tasks (redundant join processes) are detected, and merged with a global HiveQL query that answers all the incoming queries. Details of this process is explained in Chapter 4. The driver component passes the global query to the Hive compiler that produces a logical plan using information in Metastore and optimize this plan using a single rule-based optimizer. The execution engine receives a directed acyclic graph of MapReduce tasks and associated HDFS tasks and executes it in accordance with the dependencies of the tasks.

The new MQO component does not require any big changes in the system architecture of Hadoop Hive. Therefore, it can be integrated easily by Hive without requiring many modifications. Other SQL-to-MapReduce translators can take advantage of this technique as well.

The following components are the main building blocks in Hive ([42]):

- Metastore – The component that stores the system catalog and metadata about tables, columns, partitions etc.
- Driver – The component that manages the lifecycle of a HiveQL statement as it moves through Hive. The driver also maintains a session handle and any session statistics.

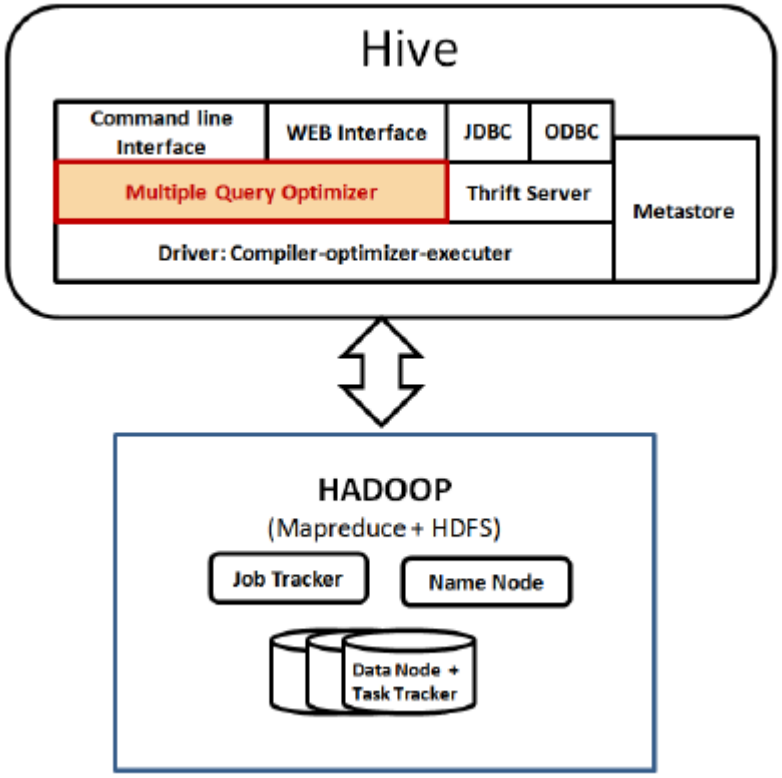

Figure 3.1: SharedHive, A novel Hadoop Hive system architecture with MQO support.

- Query Compiler – The component that compiles HiveQL into a directed acyclic graph of map/reduce tasks.
- Execution Engine – The component that executes the tasks produced by the compiler in proper dependency order. The execution engine interacts with the Hadoop instance.
- HiveServer – The component that provides a thrift interface and a JDBC/ODBC server and provides a way of integrating Hive with other applications.
- Clients components like the Command Line Interface (CLI), the web UI and JDBC/ODBC driver.
- Extensibility Interfaces which include the SerDe and ObjectInspector interfaces already described previously as well as the UDF(User Defined Function) and UDAF(User Defined Aggregate Function) interfaces that enable users to define their own custom functions.

A HiveQL statement is submitted via the CLI, the web UI or an external client using the Thrift, ODBC or JDBC interfaces [42]. The driver first passes the query to the compiler where it goes through the typical parse, type check and semantic analysis phases, using the metadata stored in the Metastore. The compiler generates a logical plan that is then optimized through a simple rule based optimizer. Finally an optimized plan in the form of a DAG of map-reduce tasks and hdfs tasks is generated. The execution engine then executes these tasks in the order of their dependencies, using Hadoop.

3.2 The Compiler Layer

The driver invokes the compiler with the HiveQL string which can be one of DDL, DML or query statements. The compiler converts the string to a plan. The plan consists only of metadata operations in case of DDL statements, and HDFS operations in case of LOAD statements. For insert statements and queries, the plan consists of a directedacyclic graph (DAG) of map-reduce jobs. ([42])

- The Parser transforms a query string to a parse tree representation.
- The Semantic Analyzer transforms the parse tree to a block-based internal query representation. It retrieves schema information of the input tables from the metastore. Using this information it veries column names, expands "select *" and does type-checking including addition of implicit type conversions.
- The Logical Plan Generator converts the internal query representation to a logical plan, which consists of a tree of logical operators.
- The Optimizer performs multiple passes over the logical plan and rewrites it in several ways:
  - Combines multiple joins which share the join key into a single multi-way join, and hence a single map-reduce job.
  - Adds repartition operators (also known as ReduceSinkOperator) for join, group-by and custom map-reduce operators. These repartition operators mark the boundary between the map phase and a reduce phase during physical plan generation.
  - Prunes columns early and pushes predicates closer to the table scan operators in order to minimize the amount of data transfered between operators.
  - In case of partitioned tables, prunes partitions that are not needed by the query
  - In case of sampling queries, prunes buckets that are not needed Users can also provide hints to the optimizer to
  - Adds partial aggregation operators to handle large cardinality grouped aggregations
  - Adds repartition operators to handle skew in grouped aggregations
  - Performs joins in the map phase instead of the reduce phase
- The Physical Plan Generator converts the logical plan into a physical plan, consisting of a DAG of mapreduce jobs. It creates a new map-reduce job for each of the marker operators – repartition and union all – in the logical plan. It then assigns portions of the logical plan enclosed between the markers to mappers and reducers of the map-reduce jobs.

In Hive, there is "query optimizer" layer for optimizing queries before converting them to map-reduce jobs and executing them. But "query optimizer" is only a single query based so there is no way to customize multiple queries in Hive with customer "query optimizer" layer.

3.3 The SharedHive Layer

In our solution, SharedHive is a layer before Hive "query optimizer". It takes multiple Hive queries and builds their execution plan by using Hive "query parser". This parser generates query plan in a tree structure. Then SharedHive sends multiple query plans from multiple Hive queries to SharedHive "optimization" layer. SharedHive "optimization" layer is a plug-in based layer. New optimization rules can be added as new plugin to this layer and they are used for optimization of queries. After "optimization" layer processed query plans, there are global query plans less than original query plans. Suppose there are "m" original query plans, there will be "n" global query plans after optimization with "n <= m" condition. Unfortunately, Hive doesn't support executing multiple queries at same time. For this reason, all optimized global queries are executed as sequential. For executing an optimized global query, its query plan is sent to Hive's "'semantic analyzer"' sub-layer of "'compiler"' layer directly by bypassing "parser" sub-layer

CHAPTER 4

MULTIPLE-QUERY OPTIMIZATION ON HIVEQL QUERIES

In this Chapter, we introduce our global query construction algorithm for HiveQL queries. We use an optimization algorithm for constructing a global HiveQL query from a set of correlated queries which produces answers for all of the input queries as HDFS files that could be used as inputs to further HiveQL queries.

4.1 The SharedHive Optimization Algorithm

Our optimization method is based on "multi-table-insert" support provided by Hive [43].

In Listing 4.1 we give a sample multi-table-insert query and next present its execution plan:

Listing 4.1: Sample Multi-Table-Insert query.

```
FROM
(
        SELECT a.status, b.school, b.gender
                FROM statusupdates a JOIN profiles b
                ON (a.userid = b.userid AND a.ds = '2009-03-20')
) subq1

        INSERT OVERWRITE TABLE gendersummary
                PARTITION(ds = '2009-03-20')
                SELECT subq1.gender, COUNT(1)
                GROUP BY subq1.gender

        INSERT OVERWRITE TABLE schoolsummary
                PARTITION(ds = '2009-03-20')
                SELECT subq1.school, COUNT(1)
                GROUP BY subq1.school;
```

This query has a single join followed by two different aggregations. By writing the query as a multi-table-insert, we make sure that the join is performed only once. The plan for the query is shown in Figure 4.1) below.

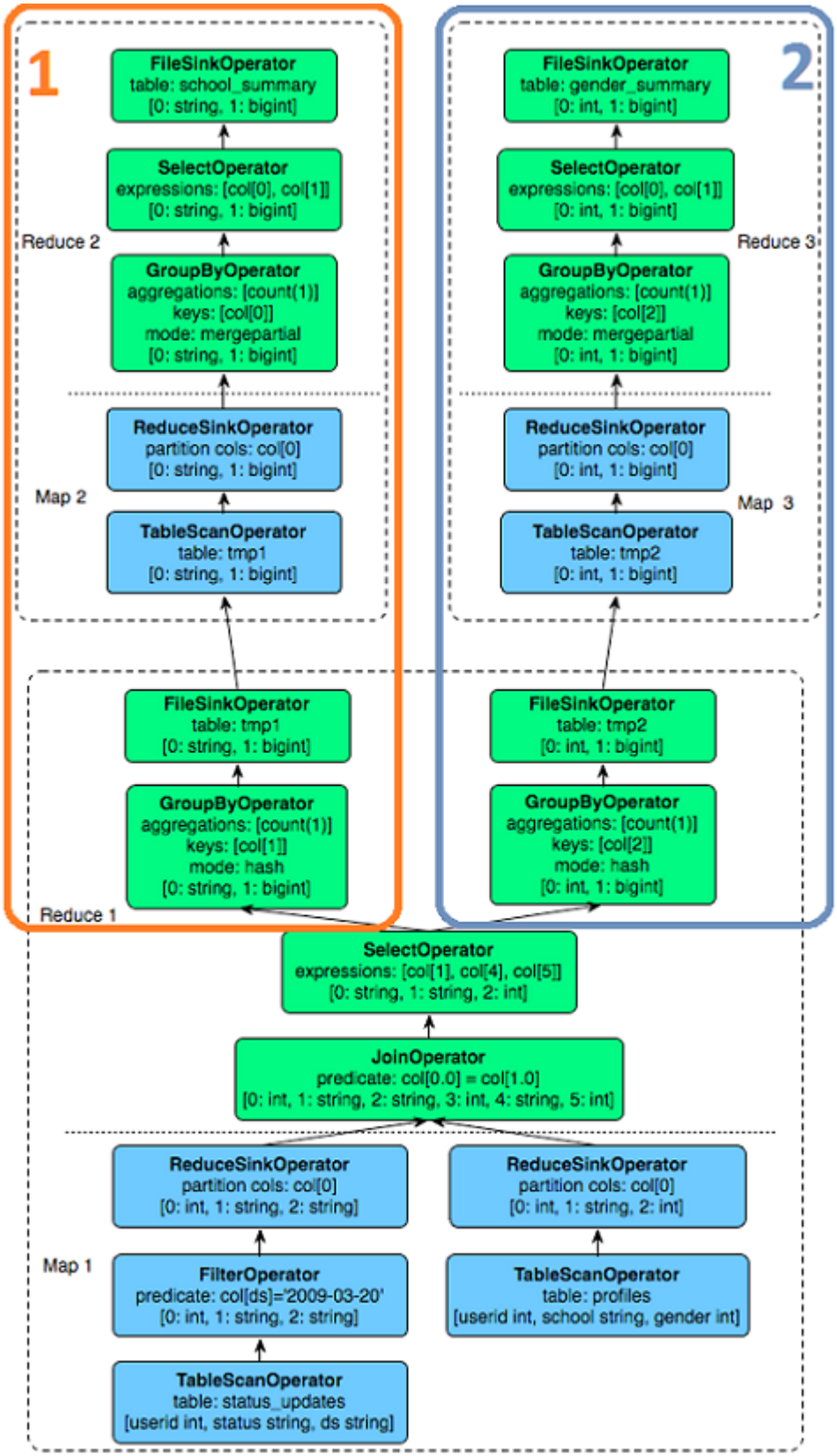

Figure 4.1: Query plan for multi-table insert query.

From Facebook Data Infrastructure Team. VLDB-2009 ([42])

The nodes in the plan are basic database operators and edges connecting nodes represent flow of data between operators. The last line in each node represents the output schema of that operator. The plan has three map-reduce jobs. Within the same map-reduce job, the portion of the operator tree below the repartition operator (ReduceSinkOperator) is executed by the mapper and the portion above is executed by the reducer. The repartitioning itself is performed by the execution engine. Notice that the first map-reduce job writes to two temporary files on HDFS, "tmp1" and "tmp2", which are consumed by the second and third map-reduce jobs, respectively. Thus, the second and third map-reduce jobs wait for the first map-reduce job to finish. ([42])

As you can see in Figure 4.1 above, Part-1 and Part-2 of query plan are independent from each other and they can be executed in parallel. We use this method in SharedHive, which detects commons parts of queries and merges intersecting queries by executing common parts only once and the other independent parts of input queries can be executed in parallel. In order to enable parallel execution in Hive, parallel execution flag must be set by *"set hive.exec.parallel=true;"*. From the viewpoint of Hive, the simple "gender_summary" query has 2 stages and "school_summary" has 2 stages also. If they are executed as sequential queries separately from each other, there will be 4 stages in total. By executing them with "multiple-table-insert" support in Hive, the number of stages is reduced to 3 as shown in Figure 4.1.

The syntax of Hive multi-table insert query is given in Listing 4.2:

Listing 4.2: Hive Multi-Table-Insert query syntax.

```
FROM from_statement
    INSERT OVERWRITE TABLE tablename_1
        [PARTITION (partcol1=val1, partcol2=val2, ...)
            [IF NOT EXISTS]] select_statement_1
    [
        INSERT OVERWRITE TABLE tablename_n
        [PARTITION ... [IF NOT EXISTS]]
        select_statement_n
    ] ...;
```

4.2 Selection of Correlated Queries

Correlated query candidates are queries having the same input table (given in the "FROM" clause). As shown in Figure 4.2, "Query-1" and "Query-3" have the same input table (same "FROM" statement) named "Table-1" and also "Query-2" and "Query-4" have the same input table (same "FROM" statement) named "Table-2". Only their data selection and conditions statements ("SELECT" and "WHERE") are different and specific to query.

Executing these queries sequentially causes two table read operations on Table-1 and Table-2 which will read all the contents of these tables twice. If SharedHive determines that Query-1, Query-2, Query-3 and Query-4 have the same data sources and these datasources are read only once, we can reduce the total execution times of these queries significantly (especially if the data common data source has a very large number of records (e.g. big data)). As shown in Figure 4.3, after their common tasks are executed only once, remaining and query specific tasks can be executed in parallel separately from each other.

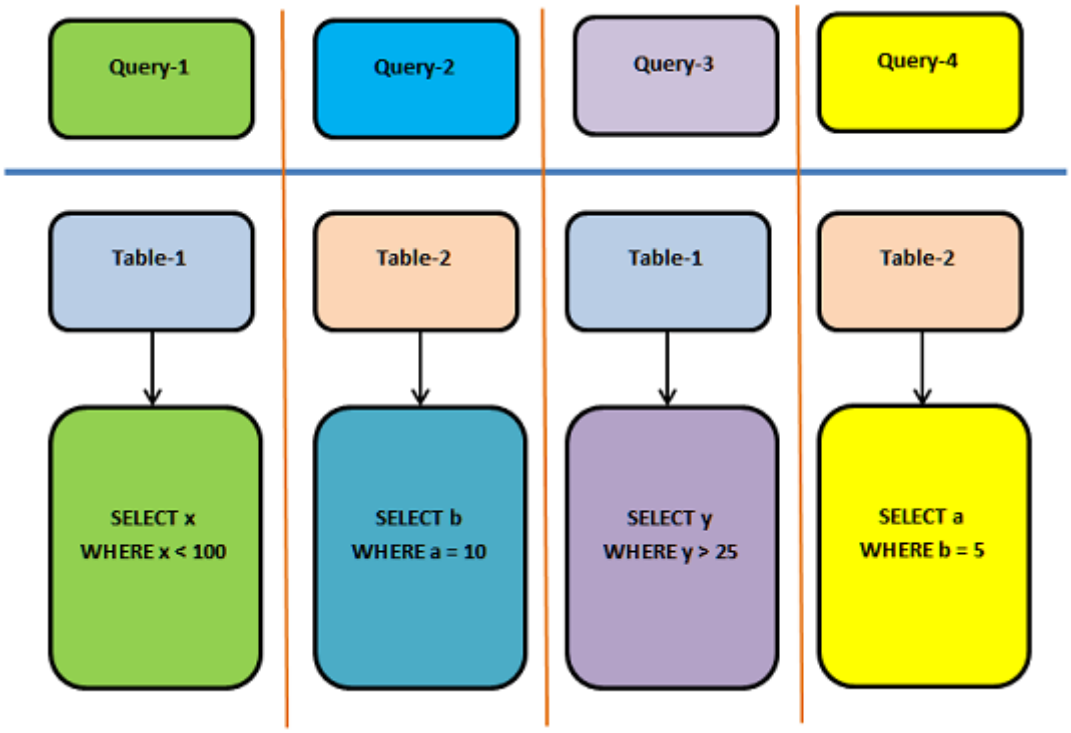

Figure 4.2: Sample correlated queries on Table-1 and Table-2.

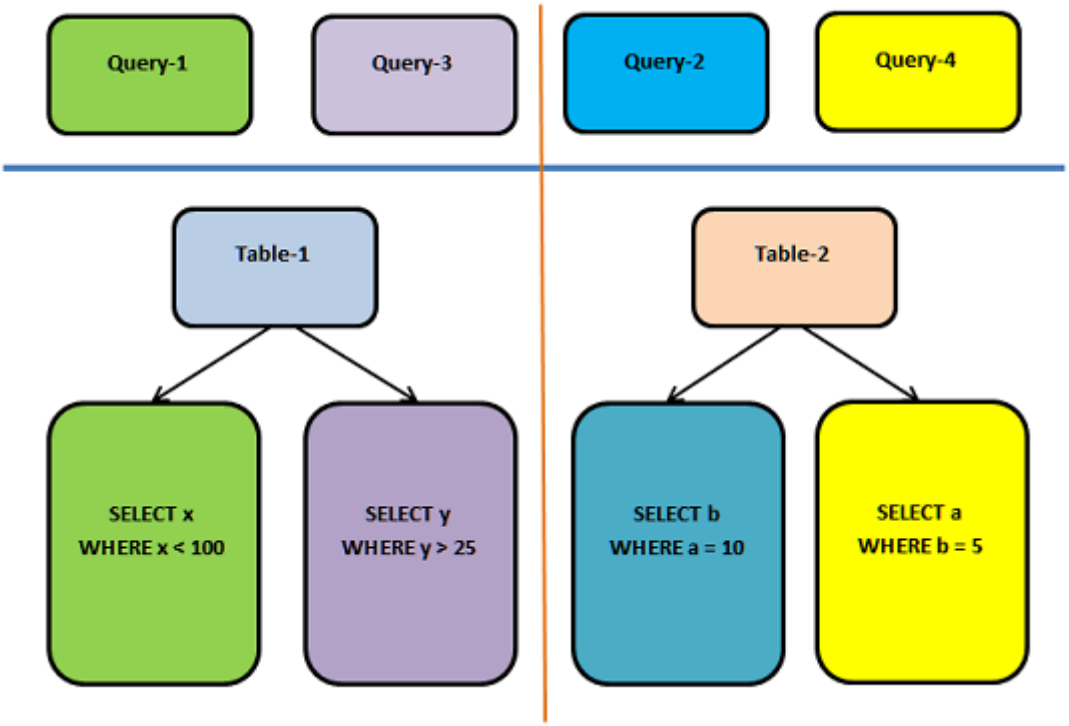

Figure 4.3: Parallel and shared execution of correlated queries.

The algorithm used in SharedHive for detecting common tasks and merging them into global queries is given below in Algorithm 1:

Algorithm 1 Generating Global HiveQL Queries.

```
Input Q_org=HiveQL (q_1 ,..., q_n);
Output Q_opt=HiveQL (q'_1 ,..., q'_m), where (m<=n);
Q_opt:=create_HiveQL_list (); //optimized query list
for q_i ∈ Q_org do
  FROM_clause=get_FROM_clause(q_i);
  MAP[FROM_clause]:=q_i ∪ map[FROM_clause];
end for
for k_i ∈ MAP.key do
  q_list:=MAP[k_i];
  if (q_list.size()==1) then
    add_query (Q_opt, q_list[0])
  else
    new_query_node = create_HiveQL (k_i);
    for (q_j ∈ q_list) do
      (q_j := q_j − k_i);
      add_query (new_query_node, q_j);
    end for
    add_query_node (Q_opt, new_query_node);
  end if
end for
```

Symbols q_i and Q denote a single query and the set of the incoming queries, respectively. The MQO formulation used by SharedHive can be given formally as:

Input: a set of queries Q_1=$\{q_1,...,q_n\}$.
Output: a set of modified queries $Q_1^{'}$=$\{q_1^{'},...,q_m^{'}\}$.
The total execution time of queries in $Q_1^{'}$ is less than Q_1.
$\sum_{i=1}^{m} execution\_time(q_i^{'}) < \sum_{i=1}^{n} execution\_time(q_i)$

If q_m is the merged query obtained by merging q_i and q_j then all of the tuples and columns required by these queries must be produced by q_m by preserving the attributes of the predicates of q_i and q_j.

The architecture of Hive cannot evaluate more than one HiveQL query and produces many jobs that run in parallel to answer the queries. The output global HiveQL query produced by SharedHive MQO component is has a valid syntax to be given as input to the Hive query optimizer and processor.

In the merging process, first we generate the query execution plans of the input queries and classify each query execution plan according to the related tables in the FROM clause of HiveQL statements. The FROM clause of HiveQL queries can have one or more relations specified in it. Therefore, our algorithm finds a global plan to execute the queries by merging them. These query plans are inserted into a tree structure that maintains the similar queries

of the parent query node. Children of this tree are the query plans that share the same tables with the parent query node. By using this technique, the input relations are not scanned redundantly while executing the child queries. This merged HiveQL query is passed to the query execution layer of Hive. The detailed explanation of the merging process is given in Algorithm 1.

The example below shows a merged global HiveQL query from two correlated TPC-H queries (Q1 and Q6). The output for each original query is written to separate files qr_1 and qr_2 after the execution of the merged query is completed which adds a minimal overhead to the global query execution time.

Merging TPC-H Queries Q1 and Q6 :

Listing 4.3: Content of "Query Q1".

```
CREATE EXTERNAL TABLE LINEITEM
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
-----------------------------------------------------
CREATE TABLE q1_pricing_summary_report
        (L RETURNFLAG STRING, ..., COUNT_ORDER INT);
-----------------------------------------------------
INSERT OVERWRITE TABLE q1_pricing_summary_report
        SELECT L_RETURNFLAG, ..., COUNT(*)
        FROM LINEITEM
        WHERE L_SHIPDATE <=  1998 09 02
        GROUP BY L_RETURNFLAG, L_LINESTATUS
        ORDER BY L_RETURNFLAG, L_LINESTATUS;
```

Listing 4.4: Content of "Query Q6".

```
CREATE EXTERNAL TABLE LINEITEM
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
-----------------------------------------------------
CREATE TABLE q6_forecast_revenue_change (REVENUE DOUBLE);
-----------------------------------------------------
INSERT OVERWRITE TABLE q6_forecast_revenue_change
        SELECT SUM(...) AS REVENUE
        FROM LINEITEM
        WHERE
                L_SHIPDATE >= '1994-01-01' AND
                L_SHIPDATE <= '1995-01-01' AND
                L_DISCOUNT >= 0.05 AND
                L_DISCOUNT <= 0.07 AND
                L_QUANTITY < 24;
```

Listing 4.5: Merged Global Query for "Q1" and "Q6".

```
CREATE EXTERNAL TABLE LINEITEM
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
-----------------------------------------------------
```

```
CREATE TABLE q1_pricing_summary_report
        (L_RETURNFLAG STRING, ..., COUNT_ORDER INT);
-----------------------------------------------------
CREATE TABLE q6_forecast_revenue_change (REVENUE DOUBLE);
-----------------------------------------------------
FROM LINEITEM

        INSERT OVERWRITE TABLE q1_pricing_summary_report
                SELECT L_RETURNFLAG, ..., COUNT(*)
                FROM LINEITEM
                WHERE L_SHIPDATE <= 1998 09 02
                GROUP BY L_RETURNFLAG, L_LINESTATUS
                ORDER BY L_RETURNFLAG, L_LINESTATUS

        INSERT OVERWRITE TABLE q6_forecast_revenue_change
                SELECT SUM(...) AS REVENUE
                WHERE
                        L_SHIPDATE >= '1994-01-01' AND
                        L_SHIPDATE <= '1995-01-01' AND
                        L_DISCOUNT >= 0.05 AND
                        L_DISCOUNT <= 0.07 AND
                        L_QUANTITY < 24;
```

In the last phase of execution, the "INSERT OVERWRITE TABLE" part of query is converted into its original form and the reducer tasks write the result of the query to its corresponding query result file, qr_i.

CHAPTER 5

EXPERIMENTAL SETUP AND RESULTS

5.1 Experimental Environment

We have run experiments on input queries 10 times and reported their averages. The observed variance in the experiments were negligible. We performed experiments with TPC-H queries adapted to HiveQL [34] and the number of submitted queries was increased up to 160 and no degradation was observed in the performance. Our experiments were performed on a High Performance Cluster (HPC) machine (using 20 nodes). The software and hardware configurations are given in Table 5.1 and the configuration of Hadoop and Hive used during the experiments is given in Table 5.2.

Table 5.1: Hardware and Software Configurations.

Hardware/Software	Model/Version
CPU	20 x 2 = 40 CPUs
Core	20 x 2 x 4 = 160 Cores
Memory	20 x 16 GB = 320 GB
Disk	20 x 146 GB = 3 TB
Network	2 x 3Com 4200G 24-port Gigabit Ethernet Switch
	1 x Voltaire 9240D 24-port Infiniband Switch
Operating System	Linux A Scientific Linux v5.2 64-bit
File System	Lustre v1.6.7 (Parallel File System)
Resource Manager	Torque v2.3.6
Job Scheduler	Maui v3.2.6
Hadoop	0.20.1
Hive	0.9.0

5.2 Query Sets

In the experiments of this thesis, we use three TPC-H query sets that each query set has two TPC-H queries (totally 6 TPC-H queries are used in the experiments) over TPC-H dataset. These queries are selected because of their potential of sharing their input tables.

Table 5.2: Default Hadoop and Hive cluster settings.

Configuration	Default value
file buffer size	4 KB
blocksize	64 MB
replication	3
namenode.handler.count	10
maximum map tasks	2
maximum reduce tasks	2
reduce tasks	1
parallel thread count	8
merge size per task	256 MB
number of nodes	20

5.2.1 TPC-H Query Set 1

5.2.1.1 Pricing Summary Report Query (Q1)

This query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included [35].

5.2.1.2 Forecasting Revenue Change Query (Q6)

This query considers all the lineitems shipped in a given year with discounts between DISCOUNT-0.01 and DISCOUNT+0.01. The query lists the amount by which the total revenue would have increased if these discounts had been eliminated for lineitems with L_QUANTITY less than quantity. Note that the potential revenue increase is equal to the sum of [L_EXTENDEDPRICE * L_DISCOUNT] for all lineitems with discounts and quantities in the qualifying range [35].

5.2.2 TPC-H Query Set 2

5.2.2.1 Promotion Effect Query (Q14)

The Promotion Effect Query determines what percentage of the revenue in a given year and month was derived from promotional parts. The query considers only parts actually shipped in that month and gives the percentage. Revenue is defined as (L_EXTENDEDPRICE*(1-L_DISCOUNT)) [35].

5.2.2.2 Discounted Revenue Query (Q19)

The Discounted Revenue query finds the gross discounted revenue for all orders for three different types of parts that were shipped by air or delivered in person. Parts are selected based on the combination of specific brands, a list of containers, and a range of sizes [35].

5.2.3 TPC-H Query Set 3

5.2.3.1 Shipping Priority Query (Q3)

The Shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of L_EXTENDEDPRICE * (1-L_DISCOUNT), of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more than 10 unshipped orders exist, only the 10 orders with the largest revenue are listed [35].

5.2.3.2 Large Volume Customer Query (Q18)

The Large Volume Customer Query ranks customers based on their having placed a large quantity order. Large quantity orders are defined as those orders whose total quantity is above a certain level. This query finds a list of the top 100 customers who have ever placed large quantity orders. The query lists the customer name, customer key, the order key, date and total price and the quantity for the order [35].

5.3 Benchmark Setup

We evaluated the performance of the Hive system with 1GB, 10GB, 25GB and 50GB standard datasets generated with the TPC-H DBGEN program. We pre-loaded the data onto the Hadoop Distributed File System (HDFS) before running all the queries. We do not consider loading time as part of the evaluated timing results. All the query results are saved into Hive tables which are also stored in HDFS. The time for storing the results into HDFS are included in the calculated timings.

5.4 System Setup for Evaluating SharedHive

In this benchmark, to support Hadoop and Hive, we have installed Hadoop on Demand (HOD) [38], a system for provisioning virtual Hadoop clusters over a large physical cluster.

We set up our Hive system on a 20 node cluster running Linux. In order to execute Hive/Hadoop system, we selected one node to be the Hadoop master/name and the rest are used as job tracker. Remaining 18 nodes are used as Hadoop slaves which execute the data nodes and task trackers. The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself. The JobTracker is the service within Hadoop that

farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack.

A TaskTracker is a node in the cluster that accepts tasks - Map, Reduce and Shuffle operations - from a JobTracker.

A DataNode stores data in its HDFS that is a functional filesystem has more than one DataNode, with data replicated across them. On startup, a DataNode connects to the NameNode and starts waiting for requests. It then responds to requests from the NameNode for filesystem operations.

5.4.1 Setup of the SharedHive System

1. Copy "shared-hive.jar" to "$HIVE_HOME/lib"
2. "cp $HIVE_HOME/bin/ext/cli.sh $HIVE_HOME/bin/ext/ shared-hive-cli.sh"
3. Open "shared-hive-cli.sh" and edit as follows:

Listing 5.1: Content of "shared-hive-cli.sh".

```
THISSERVICE=sharedHiveCli
export SERVICE_LIST="${SERVICE_LIST}${THISSERVICE}"

sharedHiveCli() {
  CLASS=tr.edu.metu.ceng.sharedhive.cli.SharedHiveCliDriver
  if $cygwin; then
    HIVELIB=`cygpath -w "$HIVE_LIB"`
  fi
  JAR=${HIVE_LIB}/shared_hive.jar
  exec $HADOOP jar $JAR $CLASS "$@"
}

sharedHiveCli_help () {
  sharedHiveCli "--help"
}
```

5.4.2 Generating TPC-H Data

1. We downloaded DBGen tool to generate TPC-H data from "http://www.tpc.org/tpch/spec/tpch_2_8_0.zip"

2. We unzipped the downloaded file and edit "makefile.suite" as follows:

Listing 5.2: Content of "makefile.suite".

```
...
CC          = gcc
DATABASE    = SQLSERVER
MACHINE     = LINUX
WORKLOAD    = TPCH
```

3. The command "make –f makefile.suite" is executed from the command line.

4. "./dbGen –s 1" is executed from the command line. (scale factor 1 means approximately 1GB of data will be generated)

5. Finally, we run "ls *.tbl" command from command line and verify that all test data files were generated successfully.

5.4.3 Uploading TPC-H Data to HDFS

a. We downloaded "https://issues.apache.org/jira/secure/attachment/12416615/TPC-H_on_Hive_2009-08-14.tar.gz"

b. The downloaded archive is unzipped to the allocated directory location.
(Say "$TPCH_HIVE_HOME")

c. We copied the data generated by DBGen (*.tbl files) to "$TPCH_HIVE_HOME/data" folder.

d. We run "$TPCH_HIVE_HOME/tpch_prepare_data.sh". This would load the DBGen generated data to HDFS.

e. Finally, the successfull loading of the generated TPC-H data on HDFS is verified by using "$HADOOP_HOME/bin/hadoopfs -ls /tpch;". The data were visible in HDFS.

5.5 Configuration

1. We set the required values for "HADOOP_CMD" and "HIVE_CMD" in "$TPCH_HIVE_HOME/benchmark.conf", "HADOOP_HOME", "HIVE_HOME" and "HADOOP_CONF_DIR=$HADOP_HOME/conf"

2. In "$TPCH_HIVE_HOME/benchmark.conf", the value "NUM_OF_TRIALS" is set to "10" giving the number of times the entire query set is to be executed.

3. The folder name is "$TPCH_HIVE_HOME/tpch" for the actual hive queries.

4. We only executed "Q1", "Q3", "Q6", "Q14", "Q18" and "Q19" in TPC-H queries for this section. So go to "$TPCH_HIVE_HOME/benchmark.conf" file and comment queries with "#" except "Q1", "Q3", "Q6", "Q14", "Q18" and "Q19" as follows:

Listing 5.3: Content of "makefile.suite".

```
#hive
HIVE_CMD="$HIVE_HOME/bin/hive  --service sharedHiveCli"

#hive tpch queries
#hive all benchmark queries
HIVE_TPCH_QUERIES_ALL=(
        "tpch/q1 pricing summary report.hive"
        #"tpch/q2_minimum_cost_supplier.hive"
```

```
        "tpch/q3_shipping_priority.hive"
        ...
        "tpch/q6_forecast_revenue_change.hive"
        #"tpch/q7_volume_shipping.hive"
        #"tpch/q8_national_marketshare.hive"
        ...
        "tpch/q14_promotion_effect.hive"
        #"tpch/q15_top_supplier.hive"
        #"tpch/q16_parts_supplier_relationship.hive"
        ...
        "tpch/q18_large_volume_customer.hive"
        "tpch/q19_discounted_revenue.hive"
        #"tpch/q20_potential_part_promotion.hive"
        ...
)
```

"Q1" and "Q6" queries are considered in TPC-H Query Set 1. "q1_pricing_summary_report.hive" and "q6_forecast_revenue_change.hive" queries can be found at "APPENDIX A" section.

"Q14" and "Q19" queries are considered in TPC-H Query Set 2. "q14_promotion_effect.hive" and "q19_discounted_revenue.hive" queries can be found at "APPENDIX B" section.

"Q3" and "Q18" queries are considered in TPC-H Query Set 3. "q3_shpping_priority.hive" and "q18_large_volume_customer.hive" queries can be found at "APPENDIX C" section.

5. No other changes are made on Hadoop or Hive configuration. The queries are executed with default Hadoop and Hive configurations. Some important Hadoop and Hive configuration properties and default values are shown at Table 5.2.

5.6 Execution

1. We run "$TPCH_HIVE_HOME/tpch_benchmark.sh" from command line.

2. The queries were executed and the statistics were printed on screen.
(and in "$TPCH_HIVE_HOME/benchmark.log")

5.7 Query Execution Results

5.7.1 All TPC-H Queries

The Hive system is evaluated with 1GB, 10GB, 25GB and 50GB standard datasets generated with the TPC-H DBGEN program ([37]). We pre-loaded the data onto the Hadoop Distributed File System (HDFS) before running all the queries. We do not consider loading time as part of the benchmark results. All the query results were saved into Hive tables which are stored in HDFS. We included those storing time in the results.

There are totally 22 queries in the TPC-H benchmark. According to the TPC-H Benchmark

requirement [36], we ran those queries for 10 times and collected their timings. The average query execution times are plotted in Figure 5.1.

The configuration of the system is summarized in Table 5.1 and also the configuration of Hadoop and Hive used during the experiments is given in Table 5.2.

In all of the experiments, the HPC system had 20 nodes and two of them are "name" and "jobtracker" nodes so there are 18 "tasktracker" nodes. Each of them can run 4 tasks (default configuration) at most. So there can be "18 x 4 = 72" parallel tasks at most. Since "Default block size" is 64MB, 64MB x 72 = 4608MB, approximately 4.5GB. Hence we can say that there can be 4.5GB data processed in parallel at most as theoretical and processing data bigger than 4.5GB can be done at multi-job cycles and these cycles are executed sequentially. This means that processing time for data bigger than 4.5GB increases almost linearly with increasing size of data. As shown in Figure 5.1, for 10GB, 25GB and 50GB datasets, there is a linear ratio between their execution times and corresponding data sizes. However, this ratio was not observed for data sizes between 1GB and 10GB. Since 4.5GB is the limit of our system only data upto 4.5GB can be processed in parallel.

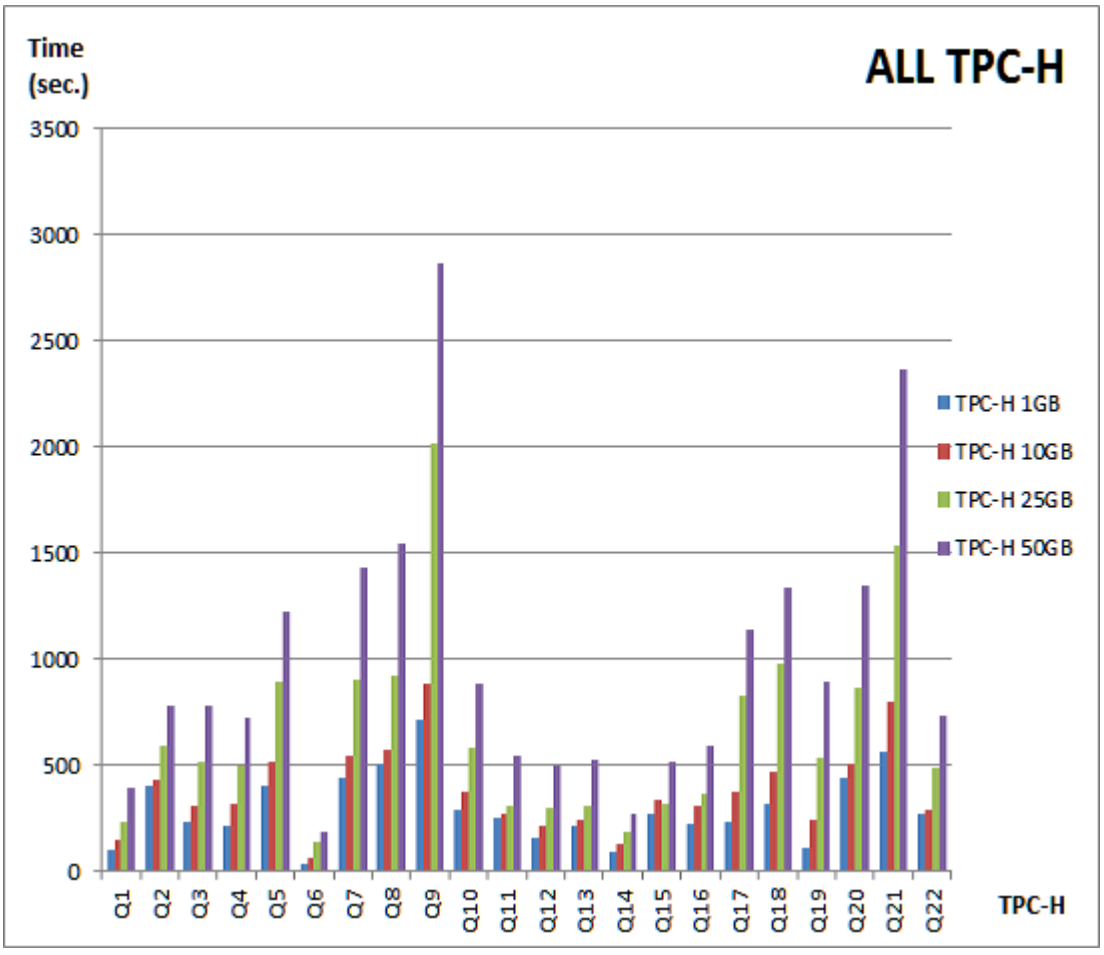

Figure 5.1: All TPC-H Queries Execution Times.

5.7.2 Q1 (Pricing Summary Report Query)

In this experiment, we generated Q1 (Pricing Summary Report Query) queries for different "L_SHIPDATE" values in "WHERE" condition. Since all generated Q1 queries have the same datasource (FROM statement such as "FROM lineitem"), they can be correlated and merged into one optimized global query. In Table 5.3, there are execution times of these generated Q1 queries for sequential execution and for parallel execution by using same datasource with different generated query counts.

Table 5.3: Q1 Execution Results.

	Non-Shared execution (sec.)				With MQO (sec.)			
Number of Queries	1GB	10GB	25GB	50GB	1GB	10GB	25GB	50GB
1	96	143	233	396	96	143	233	396
5	480	715	1,165	1,980	223	474	796	1,374
10	960	1,430	2,330	3,960	346	805	1,549	2,465
15	1,440	2,145	3,495	5,940	467	1,176	2,056	3,409
20	1,920	2,860	4,660	7,920	590	1,519	2,477	4,178
30	2,880	4,290	6,990	11,880	838	2,186	3,255	5,873
40	3,840	5,720	9,320	15,840	1,082	2,716	4,089	7,519
60	5,760	8,580	13,980	23,760	1,575	3,478	6,148	11,043
80	7,680	11,440	18,640	31,680	2,066	3,734	8,074	14,705

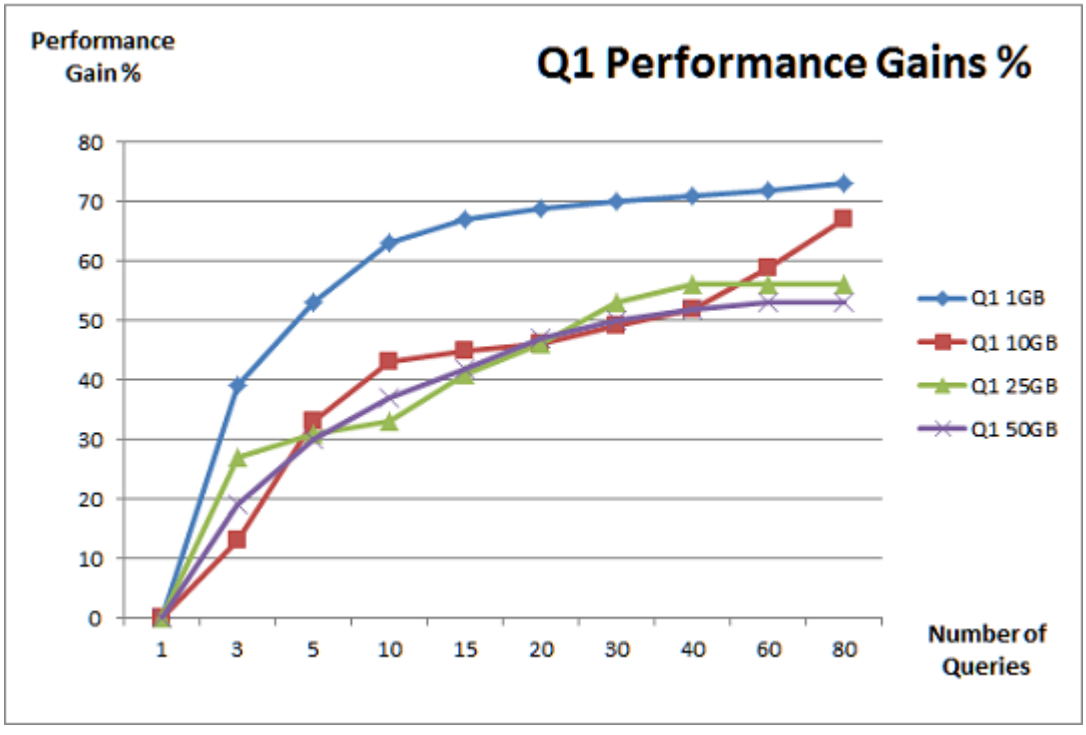

Figure 5.2: Q1 Execution Times.

5.7.3 Q1 (Pricing Summary Report Query) + Q6 (Forecasting Revenue Change Query)

In this experiment, we generated Q1 (Pricing Summary Report Query) queries for different "L_SHIPDATE" values in "WHERE" condition and generated Q6 (Forecasting Revenue Change Query) queries for different "L_SHIPDATE" values in "WHERE" condition like Q1. Since all generated Q1 and Q6 queries have the same datasource (FROM statement such as "FROM lineitem"), they can be correlated and merged into one optimized global query (6). In Table 5.4, there are execution times of these generated Q1 and Q6 queries for sequential execution and for parallel execution by using same datasource with different generated query counts. Number of queries presented in Table 5.4 and Figure 5.3, means number of Q1 and Q6 query sets (Each set has two query, Q1 and Q6).

Table 5.4: Q1+Q6 Execution Results.

	Non-Shared execution (sec.)				With MQO (sec.)			
Number of Queries	1GB	10GB	25GB	50GB	1GB	10GB	25GB	50GB
1	130	204	374	582	113	176	251	452
10	1,300	2,040	3,740	5,820	403	1,043	1,931	3,712
20	2,600	4,080	7,480	11,640	708	1,784	3,524	6,845
30	3,900	6,120	11,220	17,460	979	2,445	4,845	9,287
40	5,200	8,160	14,960	23,280	1,206	2,788	5,988	11,470
60	7,800	12,240	22,440	34,920	1,848	3,523	8,407	16,579
80	10,400	16,320	29,920	46,560	2,356	3,876	11,078	21,047

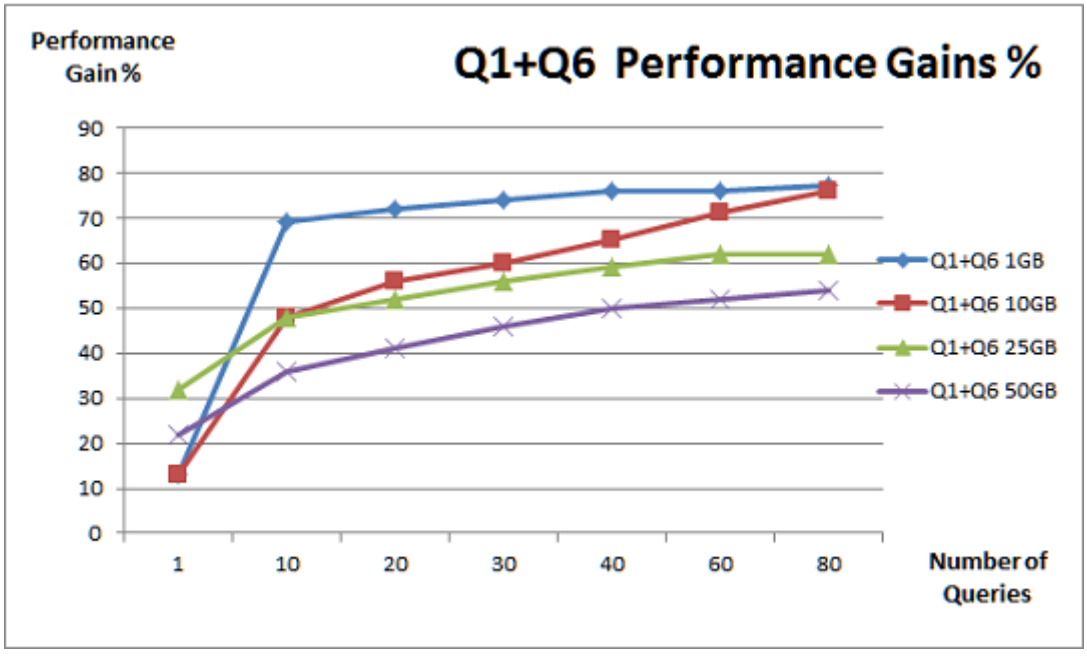

Figure 5.3: Q1+Q6 Performance Improvements.

5.7.4 Q14 (Promotion Effect Query) + Q19 (Discounted Revenue Query)

In this experiment, we generated Q14 (Promotion Effect Query) queries for different "L_SHIPDATE" values and generated Q19 (Discounted Revenue Query) queries for different "L_QUANTITY" and "P_SIZE" values in "WHERE" conditions. Because of all generated Q14 and Q19 queries have same datasource ("FROM part p JOIN lineitem l ON l.L_PARTKEY = p.P_PARTKEY"), they can be correlated and merged into one optimized global query (A.3). In Table 5.5, there are execution times of generated Q14 and Q19 queries for sequential and parallel execution. Note that joining "lineitem" and "part" tables will have a large cost because "lineitem" table includes 70 percent of all generated data. By correlating these queries, FROM statement is executed only once and so gained performance for correlated Q14 and Q19 queries are bigger than that gained from correlated Q1 and Q6 queries. Number of queries presented in Table 5.5 and Figure 5.4, means number of Q14 and Q19 query sets (Each set has two query, Q14 and Q19).

Table 5.5: Q14+Q19 Execution Results.

	Non-Shared execution (sec.)				With MQO (sec.)			
Number of Queries	1GB	10GB	25GB	50GB	1GB	10GB	25GB	50GB
1	204	364	721	1,163	161	300	608	979
10	2,040	3,640	7,210	11,630	194	331	852	1,567
20	4,080	7,280	14,420	23,260	226	411	1,367	2,249
30	6,120	10,920	21,630	34,890	262	454	1,834	3,323
40	8,160	14,560	28,840	46,520	298	477	2,049	4,032
60	12,240	21,840	43,260	69,780	363	573	2,870	5,401
80	16,320	29,120	57,680	93,040	427	603	3,342	6,390

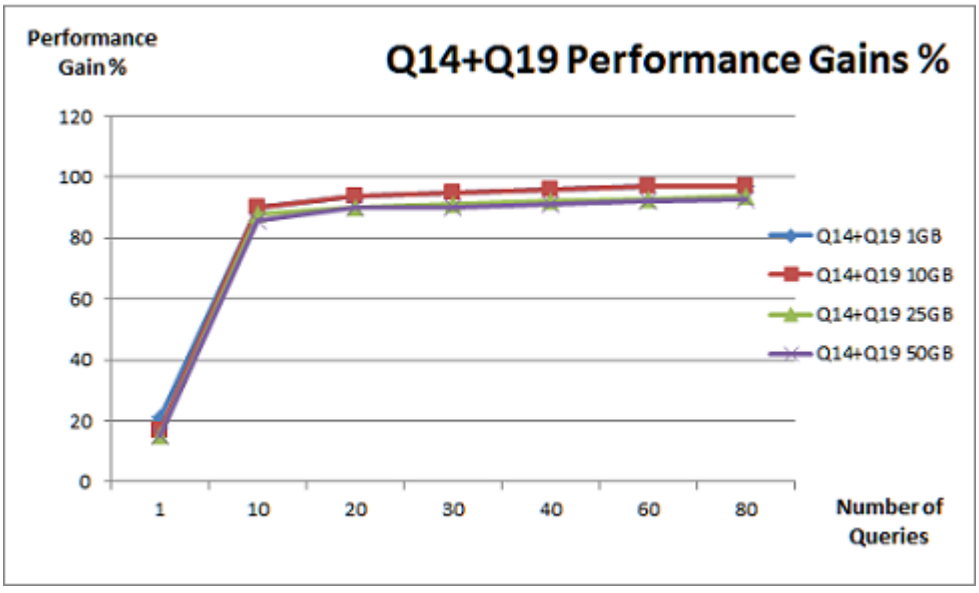

Figure 5.4: Q14+Q19 Execution Times.

5.7.5 Q3 (Shipping Priority Query) + Q18 (Large Volume Customer Query)

In this experiment, we generated Q3 (Shipping Priority Query) queries for different "L_SHIPDATE" values and generated Q18 (Large Volume Customer Query) queries for different "T_SUM_QUANTITY" values in "WHERE" conditions. But in this case, datasources (FROM statements) of Q3 and Q18 are not exactly same. Their datasources are similar. Because of SharedHive can optimize queries having same datasource, we merged these queries manually (B.3). In Table 5.6, there are execution times of generated Q3 and Q18 queries for sequential and parallel execution with different generated query counts. Note that joining "lineitem", "orders" and "customers" tables has a large cost. By correlating these queries, their costly JOIN statements are executed only once. For this reason, the performance gain for correlated Q3 and Q18 queries is bigger than that gained from Q1 and Q6 queries. Number of queries presented in Table 5.6 and Figure 5.5, means number of Q3 and Q18 query sets (Each set has two query, Q3 and Q18).

Table 5.6: Q3+Q18 Execution Results.

	Non-Shared execution (sec.)				With MQO (sec.)			
Number of Queries	1GB	10GB	25GB	50GB	1GB	10GB	25GB	50GB
1	549	780	1,491	2,119	382	620	1,313	1,891
10	5,490	7,800	14,910	21,190	1,176	1,352	2,459	4,857
20	10,980	15,600	29,820	42,380	2,062	2,416	3,347	5,806
30	16,740	23,400	44,730	63,570	2,955	3,343	4,314	6,847
40	21,960	31,200	59,640	84,760	3,845	4,106	4,622	7,831
60	32,940	46,800	89,460	127,140	5,647	5,661	5,406	10,194
80	43,920	62,400	119,280	169,520	7,426	7,512	6,891	12,263

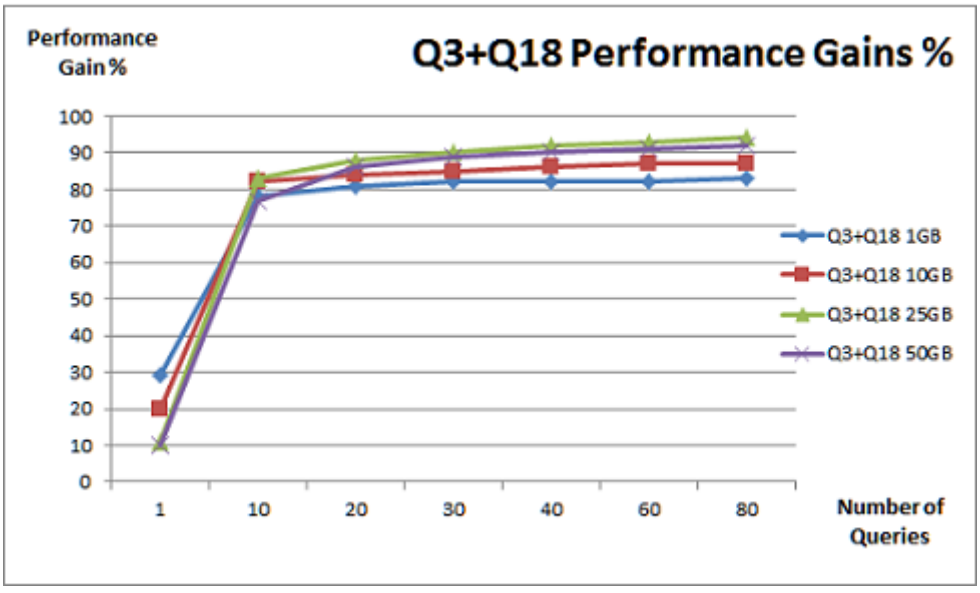

Figure 5.5: Q3+Q18 Execution Times.

5.7.6 Q1+Q6 Execution Results with Increasing Node Count

As you can see in Table 5.7, scalability of performance is observed with increased data size. Since for small data sizes, system doesn't run at full capacity and adding new nodes doesn't improve performance (i.e. scalability limit is reached). Since the system is not running at full capacity, newly added nodes are not used for processing data. Adding new nodes can gain us just a few seconds, because all TaskTracker nodes send heartbeat signals to the JobTracker node periodically and its default value is 3 seconds. By this heartbeat, JobTracker finds a new empty TaskTracker node. If we increase the number of TaskTracker nodes by adding new nodes, possibility of finding an empty TaskTracker node in less than 3 seconds is increased. This explains why execution times is reduced in 1GB data in Table 5.7 with unused new tasktracker nodes.

Table 5.7: Q1+Q6 Execution Results with Increasing Node Count.

	Execution Time (sec.)				Map Tasks / Reduce Tasks			
Number of Nodes	1GB	10GB	25GB	50GB	1GB	10GB	25GB	50GB
5	127	367	819	1,434	3/7	29/7	74/7	148/7
10	121	255	440	793	3/16	29/16	74/16	148/16
15	118	194	337	594	3/25	29/25	74/25	148/25
20	113	176	251	452	3/34	29/34	74/34	148/34

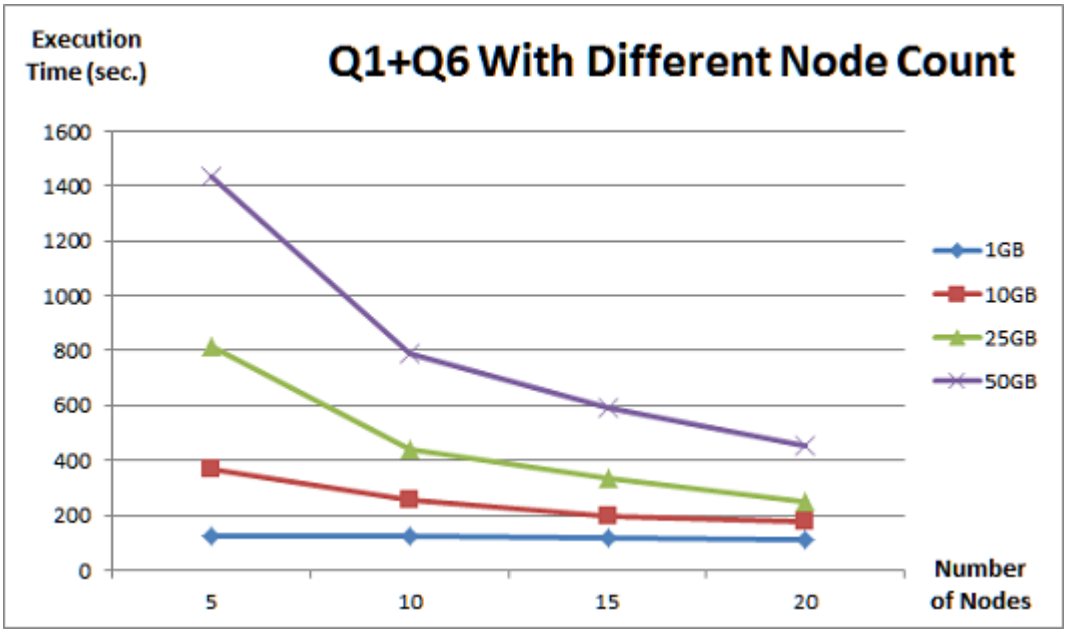

Figure 5.6: Q1+Q6 Execution Results with Increasing Node Count.

Table 5.8: Symbols and Explanations for Calculating Scalability.

Symbol	Explanation
i	Id of experiment
i-1	Id of previous experiment of experiment i
S_i	Scalability of experiment i
N_i	Number of nodes of experiment i
E_i	Execution Time of experiment i

If we calculate scalability percentage as,

$$S_i = \frac{(E_{i-1} - E_i) * 100}{E_{i-1} * (1 - \frac{N_{i-1}}{N_i})}$$

We get scalability percentages as in Figure 5.7. As shown in Figure 5.7, the more data size increases, the more scalability percentage increases. Because count of actively used tasktracker nodes increase with big data usage.

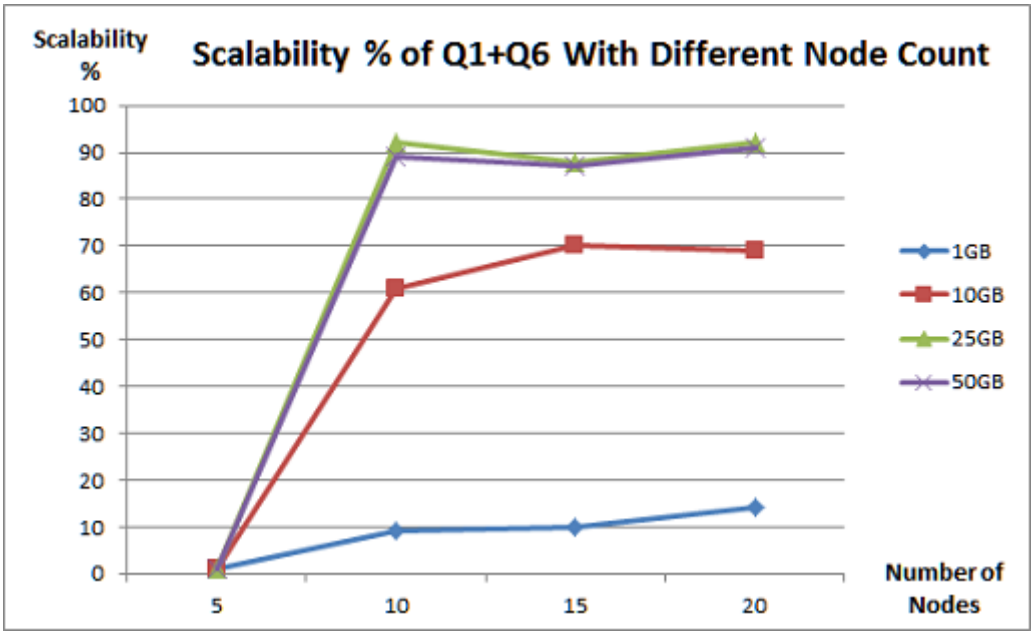

Figure 5.7: Scalability % of Q1+Q6 With Increasing Node Count.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this study, we proposed a multiple query optimization (MQO) based framework, SharedHive, to improve the performance of conventional Hadoop Hive. To our knowledge, this is the first time that the performance of Hive is being improved with MQO techniques. In SharedHive, we detected and categorized sets of correlated HiveQL queries and merged them into optimized HiveQL statements to run on Hadoop. With this approach, we showed that significant performance improvements can be achieved. Since SharedHive is designed as a new component on top of the existing Hive optimizer, it can be integrated into other SQL-to-MapReduce translators as well.

In this thesis, it is possible to process only those queries that have exactly the same datasources, not partially similar datasources. If we can detect similar common tasks (such as similar FROM statements), we can merge and optimize more TPC-H queries, increasing the potential benefits that can be achieved by SharedHive. As future work, first we plan to work on detecting correlated queries having similar datasources (FROM statements) which need not match exactly and merging them into optimized global queries.

Next, we are planning to apply MQO to the tasks in a single query. By this way, we intend to eliminate redundant tasks in queries and improve the overall performance of native rule-based query optimizer of Hive.

REFERENCES

[1] Hadoop project. http://hadoop.apache.org/.

[2] Dean, J., and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

[3] Condie, T., et al. (2010). MapReduce online. In Proceedings of the 7th USENIX conference on Networked systems design and implementation.

[4] Stonebraker, M., et al. (2010). MapReduce and parallel DBMSs: friends or foes. Communications of the ACM, 53(1), 64-71.

[5] DeWitt, D., and Stonebraker, M. (2008). MapReduce: A major step backwards. The Database Column,1.

[6] Lee, K. H., et al. (2012). Parallel data processing with MapReduce: a survey. ACM SIGMOD Record, 40(4), 11-20.

[7] Abouzeid, A., et al. (2009). HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceedings of the VLDB Endowment, 2(1), 922-933.

[8] Hive project. http://hadoop.apache.org/hive/.

[9] Ordonez, C., Song, I. Y., and Garcia-Alvarado, C. (2010). Relational versus non-relational database systems for data warehousing. In Proceedings of the ACM 13th international workshop on Data warehousing and OLAP (67-68).

[10] Thusoo, A., et al. (2010). Hive-a petabyte scale data warehouse using hadoop. ICDE,(996-1005).

[11] Sellis, T.K. (1988). Multiple-query optimization. ACM Transactions on Database Systems (TODS), 13(1), 23-52.

[12] Bayir, M. A., Toroslu, I. H., and Cosar, A. (2007). Genetic algorithm for the multiple-query optimization problem. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 37(1), 147-153.

[13] Cosar, A., Lim, E. P., and Srivastava, J. (1993). Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In Proceedings of the second international conference on Information and knowledge management (433-438).

[14] Zhou, J., Larson, P. A., Freytag, J. C., and Lehner, W. (2007). Efficient exploitation of similar subexpressions for query processing. In Proceedings of ACM SIGMOD (533-544).

[15] Chaiken, R., et al. (2008). SCOPE: easy and efficient parallel processing of massive data sets. Proceedings of the VLDB Endowment, 1(2), 1265-1276.

[16] Cohen, J., et al. (2009). MAD skills: new analysis practices for big data. VLDB, 2(2), 1481-1492.

[17] He, Y., et al. (2011). Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. ICDE (1199-1208).

[18] Lee, R., et al. (2011). Ysmart: Yet another sql-to-mapreduce translator. ICDCS (25-36).

[19] Finkelstein, S. (1982). Common expression analysis in database applications. SIGMOD (235-245).

[20] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. (2000). Efficient and extensible algorithms for multi query optimization. SIGMOD Record 29(2), 249-260.

[21] Giannikis, G., Alonso, G., and Kossmann, D. (2012). SharedDB: killing one thousand queries with one stone. Prof. of VLDB, 5(6), 526-537.

[22] Mehta, M. and DeWitt, D.J. (1995). Managing intra-operator parallelism in parallel database systems. VLDB (382-394).

[23] Beynon, M., et al. (2002). Processing large-scale multi-dimensional data in parallel and distributed environments. Parallel Computing, 28(5), 827-859.

[24] Chen, G., et al. (2011). Optimization of sub-query processing in distributed data integration systems. Journal of Network and Computer Applications, 34(4), 1035-1042.

[25] Lee, R., Zhou, M., and Liao, H. (2007). Request Window: an approach to improve throughput of RDBMS-based data integration system by utilizing data sharing across concurrent distributed queries. VLDB (1219-1230).

[26] Harizopoulos, S., Shkapenyuk, V., and Ailamaki, A. (2005). QPipe: a simultaneously pipelined relational query engine. SIGMOD (383-394).

[27] Le, W., Kementsietsidis, A., Duan, S., and Li, F. (2012). Scalable multi-query optimization for SPARQL. ICDE (666-677).

[28] Silva, Y. N., Larson, P., and Zhou, J. (2012). Exploiting Common Subexpressions for Cloud Query Processing. ICDE (1337-1348).

[29] Apache Pig. http://wiki.apache.org/pig.

[30] Bajda-Pawlikowski, K., Abadi, D. J., Silberschatz, A., and Paulson, E. (2011). Efficient processing of data warehousing queries in a split execution environment. In Proceedings of international conference on Management of data (1165-1176).

[31] Friedman, E., Pawlowski, P., and Cieslewicz, J. (2009). SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. VLDB, 2(2), 1402-1413.

[32] Nykiel, T., et al. (2010). Mrshare: Sharing across multiple queries in mapreduce. VLDB, 3(1-2), 494-505.

[33] Gruenheid, A., Omiecinski, E., and Mark, L. (2011). Query optimization using column statistics in hive. In Proceedings of the 15th Symposium on International Database Engineering and Applications (97-105).

[34] Running TPC-H queries on Hive. http://issues.apache.org/jira/browse/HIVE-600.

[35] TPC BENCHMARK H (Decision Support) Standard Specification Revision 2.3.0. http://www.tpc.org/tpch/spec/tpch2.3.0.pdf.

[36] TPC-H Benchmark Document. http://www.tpc.org/tpch/spec/tpch2.8.0.pdf.

[37] TPC-H DBGEN. http://www.tpc.org/tpch/spec/tpch_2_8_0.zip.

[38] Hadoop on Demand. http://hadoop.apache.org/docs/r0.17.0/hod.html.

[39] Introduction to Hadoop. http://developer.yahoo.com/hadoop/tutorial/module1.html

[40] Hadoop, An Elephant can't jump. But can carry heavy load. Prashant Sharma

[41] Hadoop Fundamentals: An introduction to Hive. http://fierydata.com/2012/12/03/hadoop-fundamentals-an-introduction-to-hive/

[42] Facebook Data Infrastructure Team. VLDB (2009). Hive - A Warehousing Solution Over a Map-Reduce Framework

[43] Facebook Data Infrastructure Team. ICDE (2010). Hive - A Petabyte Scale Data Warehouse Using Hadoop

[44] Hawaniah Zakaria, Shamsul Sahibuddin, Harihodin Selamat. Common Sub-Expression Identification. Proceedings of the Postgraduate Annual Research Seminar (2006)

[45] Swathi Kurunji, Tingjian Ge, Benyuan Liu, Cindy X. Chen. Communication Cost Optimization for Cloud Data Warehouse Queries. IEEE 4th International Conference on Cloud Computing Technology and Science (2012).

[46] http://en.wikipedia.org/wiki/Apache_Hadoop

[47] Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J. M., and Welton, C. (2009). MAD skills: new analysis practices for big data. Proceedings of the VLDB Endowment, 2(2), 1481-1492.

[48] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., ... and Aiyer, A. (2011). Apache Hadoop goes realtime at Facebook. In Proceedings of the 2011 international conference on Management of data (1071-1080).

[49] Capriolo, E., Wampler, D., and Rutherglen, J. (2012). Programming Hive. O'Reilly Media.

[50] http://hive.apache.org/

[51] Jarke, M. (1985). Common subexpression isolation in multiple query optimization. In Query Processing in Database Systems (pp. 191-205). Springer Berlin Heidelberg.

[52] http://developer.yahoo.com/hadoop/tutorial/module4.html

APPENDIX A

Q1 AND Q6 HIVEQL

A.1 Q1 (Pricing Summary Report Query) as HiveQL

Listing A.1: Content of "Query Q1".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
-------------------------------------------------
CREATE TABLE q1_pricing_summary_report
        (L RETURNFLAG STRING, ..., COUNT_ORDER INT);
-------------------------------------------------
INSERT OVERWRITE TABLE q1_pricing_summary_report
        SELECT L_RETURNFLAG, ..., COUNT(*)
        FROM lineitem
        WHERE L_SHIPDATE <=  1998 09 02
        GROUP BY L_RETURNFLAG, L_LINESTATUS
        ORDER BY L_RETURNFLAG, L_LINESTATUS;
```

A.2 Q6 (Forecasting Revenue Change Query) as HiveQL

Listing A.2: Content of "Query Q6".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
-------------------------------------------------
CREATE TABLE q6_forecast_revenue_change (REVENUE DOUBLE);
-------------------------------------------------
INSERT OVERWRITE TABLE q6_forecast_revenue_change
        SELECT SUM(...) AS REVENUE
        FROM lineitem
        WHERE
                L_SHIPDATE >= '1994-01-01' AND
                L_SHIPDATE <= '1995-01-01' AND
                L_DISCOUNT >= 0.05 AND
                L_DISCOUNT <= 0.07 AND
                L_QUANTITY < 24;
```

A.3 Q1+Q6 as HiveQL

Listing A.3: Merged Global Query for "Q1" and "Q6".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
-------------------------------------------------
CREATE TABLE q1_pricing_summary_report
        (L RETURNFLAG STRING, ..., COUNT_ORDER INT);
-------------------------------------------------
CREATE TABLE q6_forecast_revenue_change (REVENUE DOUBLE);
-------------------------------------------------
FROM lineitem

        INSERT OVERWRITE TABLE q1_pricing_summary_report
                SELECT L_RETURNFLAG, ..., COUNT(*)
                FROM LINEITEM
                WHERE L_SHIPDATE <=  1998 09 02
                GROUP BY L_RETURNFLAG, L_LINESTATUS
                ORDER BY L_RETURNFLAG, L_LINESTATUS

        INSERT OVERWRITE TABLE q6_forecast_revenue_change
                SELECT SUM(...) AS REVENUE
                WHERE
                        L_SHIPDATE >= '1994-01-01' AND
                        L_SHIPDATE <= '1995-01-01' AND
                        L_DISCOUNT >= 0.05 AND
                        L_DISCOUNT <= 0.07 AND
                        L_QUANTITY < 24;
```

APPENDIX B

Q14 AND Q19 HIVEQL

B.1 Q14 (Promotion Effect Query) as HiveQL

Listing B.1: Content of "Query Q14".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
CREATE EXTERNAL TABLE part
        (P_PARTKEY INT, ..., P_COMMENT STRING);
----------------------------------------------------
CREATE TABLE q14_promotion_effect (PROMO_REVENUE DOUBLE);
----------------------------------------------------
INSERT OVERWRITE TABLE q14_promotion_effect
  SELECT
    100.00
    SUM
    (
      CASE
        WHEN P_TYPE LIKE 'PROMO%'
          THEN L_EXTENDEDPRICE * (1 - L_DISCOUNT)
        ELSE 0.0
      END
    ) /
    SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS PROMO_REVENUE
  FROM part p JOIN lineitem l ON l.L_PARTKEY = p.P_PARTKEY
  WHERE l.L_SHIPDATE < '1995-10-01';
```

B.2 Q19 (Discounted Revenue Query) as HiveQL

Listing B.2: Content of "Query Q19".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
CREATE EXTERNAL TABLE part
        (P_PARTKEY INT, ..., P_COMMENT STRING);
----------------------------------------------------
CREATE TABLE q19_discounted_revenue (REVENUE DOUBLE);
```

```
-----------------------------------------------------
INSERT OVERWRITE TABLE q19_discounted_revenue
  SELECT SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE
  FROM part p JOIN lineitem l ON l.L_PARTKEY = p.P_PARTKEY
  WHERE
    (
      P_BRAND = 'Brand#12' AND
      P_CONTAINER REGEXP 'SM CASE  SM BOX  SM PACK  SM PKG' AND
      L_QUANTITY >= 1 AND
      L_QUANTITY <= 11 AND
      P_SIZE >= 1 AND
      P_SIZE <= 5 AND
      L_SHIPMODE REGEXP 'AIR  AIR REG' AND
      L_SHIPINSTRUCT = 'DELIVER IN PERSON'
    )
    OR
    (
      P_BRAND = 'Brand#23' AND
      P_CONTAINER REGEXP 'MED BAG  MED BOX  MED PKG  MED PACK' AND
      L_QUANTITY >= 10 AND
      L_QUANTITY <= 20 AND
      P_SIZE >= 1 AND
      P_SIZE <= 10 AND
      L_SHIPMODE REGEXP 'AIR  AIR REG' AND
      L_SHIPINSTRUCT = 'DELIVER IN PERSON'
    )
    OR
    (
      P_BRAND = 'Brand#34' AND
      P_CONTAINER REGEXP 'LG CASE  LG BOX  LG PACK  LG PKG' AND
      L_QUANTITY >= 20 AND
      L_QUANTITY <= 30 AND
      P_SIZE >= 1 AND
      P_SIZE <= 15 AND
      L_SHIPMODE REGEXP 'AIR  AIR REG' AND
      L_SHIPINSTRUCT = 'DELIVER IN PERSON'
    );
```

B.3 Q14+Q19 as HiveQL

Listing B.3: Merged Global Query for "Q14" and "Q19".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
CREATE EXTERNAL TABLE part
        (P_PARTKEY INT, ..., P_COMMENT STRING);
-----------------------------------------------------
CREATE TABLE q14_promotion_effect (PROMO_REVENUE DOUBLE);
```

```
CREATE TABLE q19_discounted_revenue (REVENUE DOUBLE);
--------------------------------------------------
FROM part p JOIN lineitem l ON l.L_PARTKEY = p.P_PARTKEY

  SELECT
    100.00
    SUM
    (
      CASE
        WHEN P_TYPE LIKE 'PROMO%'
          THEN L_EXTENDEDPRICE * (1 - L_DISCOUNT)
        ELSE 0.0
      END
    ) /
    SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS PROMO_REVENUE
  WHERE l.L_SHIPDATE < '1995-10-01'

        SELECT SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE
        WHERE
    (
      P_BRAND = 'Brand#12' AND
      P_CONTAINER REGEXP 'SM CASE  SM BOX  SM PACK  SM PKG' AND
      L_QUANTITY >= 1 AND
      L_QUANTITY <= 11 AND
      P_SIZE >= 1 AND
      P_SIZE <= 5 AND
      L_SHIPMODE REGEXP 'AIR  AIR REG' AND
      L_SHIPINSTRUCT = 'DELIVER IN PERSON'
    )
    OR
    (
      P_BRAND = 'Brand#23' AND
      P_CONTAINER REGEXP 'MED BAG  MED BOX  MED PKG  MED PACK' AND
      L_QUANTITY >= 10 AND
      L_QUANTITY <= 20 AND
      P_SIZE >= 1 AND
      P_SIZE <= 10 AND
      L_SHIPMODE REGEXP 'AIR  AIR REG' AND
      L_SHIPINSTRUCT = 'DELIVER IN PERSON'
    )
    OR
    (
      P_BRAND = 'Brand#34' AND
      P_CONTAINER REGEXP 'LG CASE  LG BOX  LG PACK  LG PKG' AND
      L_QUANTITY >= 20 AND
      L_QUANTITY <= 30 AND
      P_SIZE >= 1 AND
      P_SIZE <= 15 AND
      L_SHIPMODE REGEXP 'AIR  AIR REG' AND
```

```
      L_SHIPINSTRUCT = 'DELIVER IN PERSON'
    );
```

APPENDIX C

Q3 AND Q18 HIVEQL

C.1 Q3 (Shipping Priority Query) as HiveQL

Listing C.1: Content of "Query Q3".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
CREATE EXTERNAL TABLE orders
        (O_ORDERKEY INT, ..., O_COMMENT STRING);
CREATE EXTERNAL TABLE customer
        (C_CUSTKEY INT, ..., C_COMMENT STRING);
-----------------------------------------------------
CREATE TABLE q3_shipping_priority
  (
    L_ORDERKEY INT,
    REVENUE DOUBLE,
    O_ORDERDATE STRING,
    O_SHIPPRIORITY INT
  );
-----------------------------------------------------
INSERT OVERWRITE TABLE q3_shipping_priority
  SELECT
    L_ORDERKEY,
    SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE,
    O_ORDERDATE,
    O_SHIPPRIORITY
  FROM
    customer c
    JOIN
    orders o ON c.C_CUSTKEY = o.O_CUSTKEY
    JOIN
    lineitem l ON l.L_ORDERKEY = o.O_ORDERKEY
  WHERE
    C_MKTSEGMENT = 'BUILDING' AND
    O_ORDERDATE < '1995-03-15' AND
    L_SHIPDATE > '1995-03-15'
  GROUP BY
    L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
```

```
  ORDER BY
    REVENUE DESC, O_ORDERDATE
  LIMIT 10;
```

C.2 Q18 (Large Volume Customer Query) as HiveQL

Listing C.2: Content of "Query Q18".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
CREATE EXTERNAL TABLE orders
        (O_ORDERKEY INT, ..., O_COMMENT STRING);
CREATE EXTERNAL TABLE customer
        (C_CUSTKEY INT, ..., C_COMMENT STRING);
-----------------------------------------------------
CREATE TABLE q18_tmp (L_ORDERKEY INT, T_SUM_quantity DOUBLE);
CREATE TABLE q18_large_volume_customer
        (
    C_NAME STRING,
    C_CUSTKEY INT,
    O_ORDERKEY INT,
    O_ORDERDATE STRING,
    O_TOTALPRICE DOUBLE,
    SUM_QUANTITY DOUBLE
  );
-----------------------------------------------------
INSERT OVERWRITE TABLE q18_large_volume_customer
  SELECT
    C_NAME,
    C_CUSTKEY,
    O_ORDERKEY,
    O_ORDERDATE,
    O_TOTALPRICE,
    SUM(L_QUANTITY)
  FROM
    customer c
    JOIN
    orders o ON c.C_CUSTKEY = o.O_CUSTKEY
    JOIN
    lineitem l ON l.L_ORDERKEY = o.O_ORDERKEY
    JOIN
    q18_tmp t ON t.T_ORDERKEY = o.O_ORDERKEY
  WHERE t.T_SUM_QUANTITY > 300
  GROUP BY
    C_NAME,
    C_CUSTKEY,
    O_ORDERKEY,
    O_ORDERDATE,
```

```
    O_TOTALPRICE
  ORDER BY O_TOTALPRICE DESC, O_ORDERDATE
  LIMIT 100;
```

C.3 Q3+Q18 as HiveQL

Listing C.3: Merged Global Query for "Q3" and "Q18".

```
CREATE EXTERNAL TABLE lineitem
        (L_ORDERKEY INT, ..., L_COMMENT STRING);
CREATE EXTERNAL TABLE orders
        (O_ORDERKEY INT, ..., O_COMMENT STRING);
CREATE EXTERNAL TABLE customer
        (C_CUSTKEY INT, ..., C_COMMENT STRING);
--------------------------------------------------
CREATE TABLE q3_shipping_priority
  (
    L_ORDERKEY INT,
    REVENUE DOUBLE,
    O_ORDERDATE STRING,
    O_SHIPPRIORITY INT
  );
CREATE TABLE q18_tmp (L_ORDERKEY INT, T_SUM_quantity DOUBLE);
CREATE TABLE q18_large_volume_customer
  (
    C_NAME STRING,
    C_CUSTKEY INT,
    O_ORDERKEY INT,
    O_ORDERDATE STRING,
    O_TOTALPRICE DOUBLE,
    SUM_QUANTITY DOUBLE
  );
--------------------------------------------------
FROM
  customer c
  JOIN
  orders o ON c.C_CUSTKEY = o.O_CUSTKEY
  JOIN
  lineitem l ON l.L_ORDERKEY = o.O_ORDERKEY
  JOIN
  (
    SELECT
      L_ORDERKEY AS T_ORDERKEY,
      SUM(L_QUANTITY) AS T_SUM_QUANTITY
    FROM lineitem
    GROUP BY L_ORDERKEY
  ) t ON t.T_ORDERKEY = o.O_ORDERKEY
```

```
INSERT OVERWRITE TABLE q3_shipping_priority
  SELECT
    L ORDERKEY,
    SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE,
    O_ORDERDATE,
    O_SHIPPRIORITY
  WHERE
    C_MKTSEGMENT = 'BUILDING' AND
    O_ORDERDATE < '1995-03-15' AND
    L_SHIPDATE > '1995-03-15'
  GROUP BY
    L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
  ORDER BY
    REVENUE DESC, O_ORDERDATE
  LIMIT 10

INSERT OVERWRITE TABLE q18_large_volume_customer
  SELECT
    C_NAME,
    C_CUSTKEY,
    O_ORDERKEY,
    O_ORDERDATE,
    O_TOTALPRICE,
    SUM(L_QUANTITY)
  WHERE T_SUM_QUANTITY > 300
  GROUP BY
    C_NAME,
    C_CUSTKEY,
    O_ORDERKEY,
    O_ORDERDATE,
    O_TOTALPRICE
  ORDER BY O_TOTALPRICE DESC, O_ORDERDATE
  LIMIT 100;
```

VITA

PERSONAL INFORMATION

Surname, Name:	Özal, Serkan
Nationality:	Turkish (TC)
Date and Place of Birth:	15 September 1986, Çankırı/Turkey
Marital Status:	Single
Phone:	+90 312 476 93 98
GSM:	+90 542 680 39 18
E-Mail:	serkan.ozal@metu.edu.tr

EDUCATION

Degree	Institution	Year of Graduation
B.S.	Hacettepe University - Computer Engineering	2009
High School	Kastamonu Mustafa Kaya Anatolian High School	2004

WORK EXPERIENCE

Year	Place	Enrollment
2009-2010	TÜBİTAK - UEKAE / G222	Software Developer
2010-2011	MilSOFT	Software Developer
2011-	T2 Yazılım	Senior Software Developer

FOREIGN LANGUAGES

Advanced English