# University of Cincinnati

**Date: 5/30/2014**

**I, Krishna Karthik Gadiraju , hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.**

It is entitled:

**Benchmarking Performance for Migrating a Relational Application to a Parallel Implementation**

Student's name: **Krishna Karthik Gadiraju**

This work and its defense approved by:

**Committee chair: Karen DavisPh.D.**

**Committee member: Prabir BhattacharyaPh.D.**

**Committee member: Paul Talaga**

10551

# Benchmarking Performance for Migrating a Relational Application to a Parallel Implementation

A thesis submitted to the

Division of Graduate Studies and Research
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of Electrical Engineering and Computing Systems
of the College of Engineering and Applied Science

May 30, 2014

by

Krishna Karthik Gadiraju

BTech, Jawaharlal Nehru Technological University, Hyderabad, India, 2011

Thesis Advisor and Committee Chair: Dr. Karen C. Davis

**Abstract**

Many organizations rely on relational database platforms for OLAP-style querying (aggregation and filtering) for small to medium size applications. We investigate the impact of scaling up the data sizes for such queries. We intend to illustrate what kind of performance results an organization could expect should they migrate current applications to big data environments. This thesis benchmarks the performance of Hive [TSJS09], a parallel data warehouse platform that is a part of the Hadoop software stack. We set up a 4-node Hadoop cluster using Hortonworks HDP 1.3.2 [HHDP]. We use the data generator provided by the TPC-DS benchmark [TPCDS] to generate data of different scales. We use a representative query provided in the TPC-DS query set and run the SQL and Hive Query Language (HiveQL) versions of the same query on a relational database installation (MySQL) and on the Hive cluster. An analysis of the results shows that for all the dataset sizes used, Hive is faster than MySQL when executing the query. Hive loads the large datasets faster than MySQL, while it is marginally slower than MySQL when loading the smaller datasets.

## Table of Contents

List of Figures ..... v
List of Tables ..... vi
Chapter 1: Introduction ..... 1
1.1 General Research Objective ..... 1
1.2 Specific Research Objective ..... 1
1.3 Research Methodology ..... 2
1.4 Contributions of the Research ..... 2
1.5 Overview ..... 3
Chapter 2: An Overview of Hadoop and Big Data Benchmarking ..... 4
2.1 Hadoop ..... 4
2.2 Big Data Benchmarking ..... 16
Chapter 3: Hardware and Software Settings ..... 18
3.1 Hardware Configuration ..... 19
3.2 Software Configuration ..... 20
3.3 CSHadoop Architecture ..... 20
3.4 TPC-DS Schema and Features of Query 7 ..... 22
Chapter 4: Experimental Setup and Results ..... 26
4.1 Experimental Procedure ..... 26
4.2 Results ..... 31
4.3 Observations ..... 32
4.4 Discussion ..... 34
4.5 Conclusion ..... 40
Chapter 5: Contributions and Future Work ..... 41
5.1 Contributions ..... 41
5.2 Future Work ..... 42
References ..... 45
Appendix A: Instructions for Generating Data Using TPC-DS ..... 48
Appendix B: Additional Results ..... 51

## List of Figures

Figure 2.1 Hadoop system architecture....5
Figure 2.2 Execution of a program written in MapReduce....8
Figure 2.3 Hive architecture....11
Figure 2.4 Order of execution of the Hive compiler....12
Figure 2.5 Query execution plan of an example Hive query....14
Figure 3.1 CSHadoop architecture....21
Figure 3.2 An excerpt of TPC-DS snowflake schema....23
Figure 4.1 Original version of Query 7....28
Figure 4.2 Modified SQL version of Query 7....29
Figure 4.3 Differences between original and HiveQL versions of Query 7....29
Figure 4.4 HiveQL version of modified SQL query from Figure 4.2....29
Figure 4.5 A comparison of data load times for promotion dataset between MySQL and Hive....36
Figure 4.6 A comparison of data load times for item dataset between MySQL and Hive....36
Figure 4.7 A comparison of data load times for store_sales dataset between MySQL and Hive....37
Figure 4.8 A comparison of query execution time for Query between MySQL and Hive....37
Figure 4.9 Scalability comparison between MySQL and Hive for executing Query 7....38

## List of Tables

Table 2.1: A comparison of features between Hive, HBase and Pig ..... 9
Table 2.2 Differences between Hive and a traditional relational database ..... 15
Table 3.1 CSHadoop and MySQL machine hardware configuration ..... 19
Table 3.2 Cloudgate hardware configuration ..... 19
Table 3.3 Network configuration for CSHadoop and MySQL machine ..... 19
Table 3.4 Hadoop software versions installed in CSHadoop ..... 20
Table 3.5 CSHadoop cluster configuration ..... 21
Table 3.6 Features of Query 7 ..... 22
Table 3.7 Schema of different tables used in Query 7 ..... 23
Table 4.1 item table definition in SQL and HiveQL ..... 27
Table 4.2 SQL and HiveQL commands to load the data for store_sales table into MySQL and Hive respectively ..... 28
Table 4.3.1 Total size and number of records in datasets used for the query–Dataset 1 ..... 31
Table 4.3.2 Table definition, data loading and query execution times–Dataset 1 ..... 31
Table 4.4.1 Total size and number of records in datasets used for the query–Dataset 2 ..... 31
Table 4.4.2 Table definition, data loading and query execution times–Dataset 2 ..... 31
Table 4.5.1 Total size and number of records in datasets used for the query–Dataset 3 ..... 32
Table 4.5.2 Table definition, data loading and query execution times–Dataset 3 ..... 32
Table 4.6 Comparison of current study with related studies ..... 39
Table 4.7.1 Total size and number of records in datasets used for the query–Dataset 4 ..... 51
Table 4.7.2 Table definition, data loading and query execution times-Dataset 4 ..... 51
Table 4.8.1 Total size and number of records in datasets used for the query–Dataset 5 ..... 51
Table 4.8.2 Table definition, data loading and query execution times-Dataset 5 ..... 52
Table 4.9.1 Total size and number of records in datasets used for the query–Dataset 6 ..... 52
Table 4.9.2 Table definition, data loading and query execution times-Dataset 6 ..... 52
Table 4.10.1 Total size and number of records in datasets used for the query–Dataset 7 ..... 53
Table 4.10.2 Table definition, data loading and query execution times-Dataset 7 ..... 53
Table 4.11.1 Total size and number of records in datasets used for the query–Dataset 8 ..... 53
Table 4.11.2 Table definition, data loading and query execution times-Dataset 8 ..... 54

# Chapter 1: Introduction

As a result of the ever increasing reach of the internet, small and medium size businesses are now able to cater to a larger client base. Many of these organizations use relational database systems to run OLAP-style queries (aggregation and filtering) for analyzing their data. In this thesis, we investigate the impact of scaling up the data size for the aforementioned OLAP-style queries. Baru et al. [BBNP13] indicate that enterprise data is estimated to grow from 0.5 ZB in 2008 to 35 ZB in 2020. This large scale data, which comprises of both structured and unstructured components, is referred to as Big Data. The aim of this thesis is to select a parallel data management system and compare its OLAP-style query performance for large scale relational data against a relational database management system.

A parallel data management system uses distributed methods to store, manage and analyze the data. The datasets are broken down into smaller blocks and are distributed across several nodes. A query written on such a system would then run in parallel on all the smaller blocks and display the aggregated results to the user.

## 1.1 General Research Objective

The general research objective is to compare the performance of a massively parallel implementation to that of a traditional relational database using a standard benchmark.

## 1.2 Specific Research Objectives

To identify and evaluate a technology that is capable of performing petabyte scale analysis, we define the following specific research objectives:

A. Identify and set up a parallel computing platform for a parallel database system.
B. Identify parallel data management systems that run on the platform and investigate their strengths and features and select one to use for our study.
C. Select a relational database management system.

D. Select a dataset and define a set of queries to run both on the parallel and the relational data management systems.

E. Compare the performance of the two data management systems for both loading data and executing a representative query.

### 1.3 Research Methodology

In order to achieve the objectives outlined in the previous section, the following activities are conducted.

A. Survey the literature for a parallel computing platform. Investigate the features of Hadoop and the Map Reduce programming platform [DG08] it runs on.

B. Survey the literature for parallel data management systems that run on Hadoop. We investigate the features of Apache Hive [TSJS09] and Apache HBase [W12]. We compare the features of both technologies and select one technology to compare against a relational database.

C. Identify a benchmarking standard that supports large scale relational data and defines queries of varying complexity.

D. Run the queries on both the relational and parallel data management systems while varying parameters such as the number of records and number of parallel nodes.

E. Analyze the performance of both of the data management systems using the results obtained from loading data and running the queries.

### 1.4 Contributions of the Research

Successful completion of the aforementioned tasks is expected to contribute the following:

A. An overview of features of the Hadoop platform and the advantages it has for parallel computing.

B. A survey of different data management systems available on Hadoop and a comparison of their features.

C. An analysis of the performance of the parallel and relational data management systems based on a common set of queries and data.

### 1.5 Overview

In Chapter 2, we describe the features of different software packages that fall under the umbrella of the Hadoop project. We give an overview of MapReduce, a parallel computing platform and Hive, a distributed data warehouse that runs on top of MapReduce. In Chapter 3, we describe the procedure followed to set up the Hadoop cluster used in this thesis. We also describe the features of the TPC-DS benchmark which was used for generating the data and queries necessary for the experimental procedure described in Chapter 4. In Chapter 4, we also show the results obtained from the experimental procedure followed and analyze the results. In Chapter 5, we summarize the contributions to research made by this thesis and suggest future work.

# Chapter 2: An Overview of Hadoop and Big Data Benchmarking

## 2.1 Hadoop

Hadoop is a term used for a family of related software projects that fall under the umbrella of infrastructure for distributed computing and parallel processing [W12]. A Hadoop ecosystem may have the following technologies in it:

- **HDFS**, a distributed file system for Hadoop.
- **MapReduce**, a parallel programming model.
- **Apache Hive**, a data warehouse built on top of the MapReduce programming framework.
- **Pig**, a platform for analyzing large datasets, using Pig Latin, a high level programming language.
- **Sqoop**, a tool built for transferring data between Hadoop and other data sources such as relational databases.
- **HBase,** a NoSQL based distributed column store.

In our work, we use the Apache Hive data warehouse. In the following sections, we describe the architecture of a Hadoop ecosystem and elaborate on the features of those tools that are relevant to this thesis.

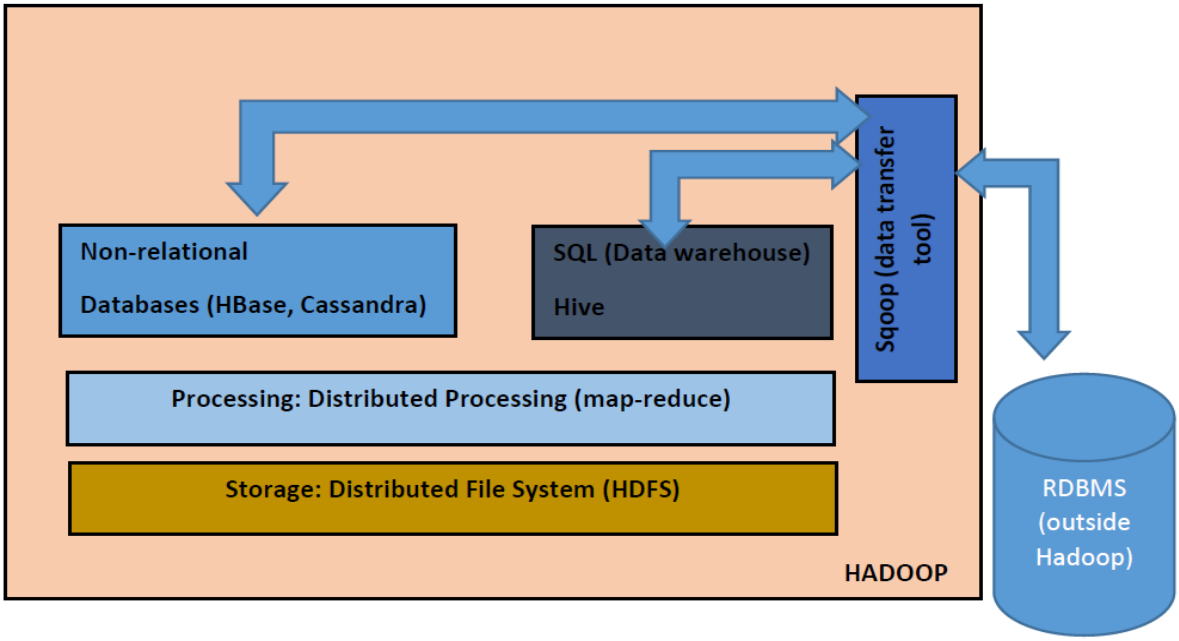
### 2.1.1 Hadoop Architecture

Figure 2.1 shows a simple representation of the architecture of the Hadoop ecosystem. Some of the popular components of Hadoop are explained in detail in this section.

**a. HDFS**

HDFS (Hadoop Distributed File System) is a distributed file system for Hadoop. Files are stored in the form of blocks in HDFS. Each block is by default 64MB in size. HDFS has a namenode and several datanodes. The namenode is responsible for maintenance and management of the entire distributed file system. It stores the information regarding all the directories and files stored in the system. It also stores the information regarding the datanodes on which the different blocks of a file are stored [W12]. A datanode is the secondary node in which the actual data blocks are stored.

**Figure 2.1** *Hadoop system architecture*




**b. MapReduce**

MapReduce is a parallel programming model and its implementation is used for processing and generating large datasets [DG08]. MapReduce was developed by Google in 2004 [DG08]. A MapReduce program typically involves users specifying a *map* function that takes *(key, value)* pairs as input, processes them and generates intermediate (key, value) pairs. A *reduce* function then groups together all the intermediate (key, value) pairs with the same key. The user can specify how these related (key, value) pairs can then be processed.

Figure 2.2 gives a schematic diagram of the order of execution of a MapReduce program [DG08]. When the user executes the MapReduce program, multiple copies of the program are created. One copy is assigned as the master and the rest as workers. The input data is split up into multiple chunks and these chunks are passed to the workers which run the map function, which converts the initial (key, value) pairs into intermediate (key, value) pairs. These intermediate (key, value) pairs are periodically written onto a local disk for retrieval by the workers running the reduce function. The workers running the reduce function then collect all the intermediate data from the local disks and group them together based on key value. All the pairs with the same key are

assigned to the same reducer which then performs the processing on these pairs as defined by the user. The outputs of each of the reducers are then written to the output files, as shown in Figure 2.2. The functioning of each individual map and reduce functions is explained below in a word count example [DG08].

Consider a MapReduce program to count the frequency of each word present in a set of documents. The program primarily contains two functions. The pseudocode for each of the functions is as shown below [DG08].

**map function [DG08]:**

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
    EmitIntermediate(w, "1");
```

The map function takes the file and its text as an input and emits each word in the file as a key and the value 1 for every occurrence of the word in the file. This (key, value) pairs are the intermediate values mentioned in Figure 2.2. In other words, the map function emits a new intermediate (key, value) pair for every word in the documents.

The MapReduce programming paradigm then collects all the intermediate (key, value) pairs. All the intermediate pairs with the same key are then grouped together and sent to a reducer. In other words, the MapReduce programming paradigm generates a reduce function for each group of intermediate (key, value) pairs. The pseudocode for the reducer function is as shown below [DG08]:

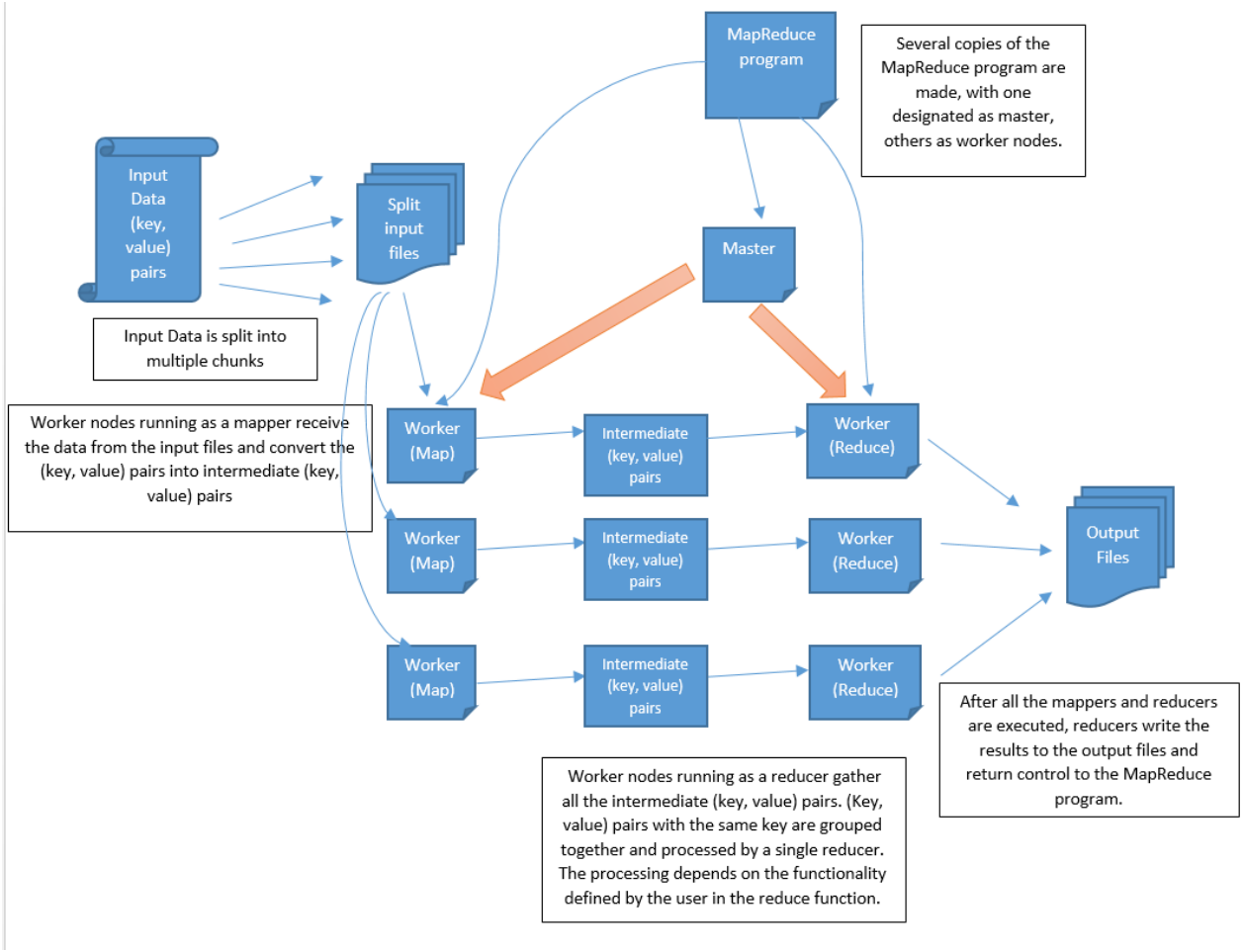
**reduce function [DG08]:**

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
    result += ParseInt(v);
Emit(AsString(result));
```

The reducer function receives all the intermediate (key, value) pairs with the same key and increments a counter for every occurrence of the intermediate key (in this example, the word). Once all the mappers and reducers have finished their processing, the results are then output onto the screen.

Stonebraker and Dewitt criticize MapReduce as a major step backward in building a programming paradigm for large scale data intensive applications [SD08]. They argue that MapReduce is not novel, is brute-force, does not define a schema, and does not support databases [SD08]. Jorgensen [J08] rejects their view by explaining that Stonebraker and Dewitt compared MapReduce to a database, while in reality, MapReduce was never designed to be a database, but more as a platform that forms a part of the databases [J08]. Since then, several data management systems/data warehouses such as Hive, Pig, and HBase have been developed; they incorporate MapReduce into their systems and answer some of the issues mentioned by Stonebraker and Dewitt such as the ability to define a schema, and a high level language to store and manage data.

**Figure 2.2** *Execution of a program written in MapReduce [DG08]*




### c. Data Management Systems on Hadoop

Some of the data management systems present on top of the Hadoop environment are Apache Hive, HBase, and Pig. Table 2.1 shows a comparison of the features of the three data management systems. Table 2.1 describes some of the main differences between Hive, HBase and Pig such as the type of language used by the three systems, their support to relational and non-relational data and their data models. While Pig supports relational and non-relational data, HBase is built to support non-relational data and Hive supports relational data. All the three data management systems use different types of languages, as described in Table 2.1. While Hive uses HiveQL (Hive Query Language), a language that is similar to SQL, Pig uses PigLatin and HBase is accessed programmatically by Java, THRIFT APIs or by using JRuby scripts. Because

of its support of relational data and HiveQL, we use Apache Hive as the big data management system in this thesis.

**Table 2.1:** *A comparison of features between Hive, HBase and Pig*

| Feature | Hive | HBase | Pig |
|---|---|---|---|
| Type | Data warehouse | Distributed column store | Data flow based platform |
| Language | HiveQL, a language similar to SQL | HBase can be programmatically accessed through the Java, REST or THRIFT APIs. We can use JRuby to write scripts. | Pig Latin, a data flow based programming language |
| Type of language | HiveQL is a declarative language | Uses different languages to access non-relational data | Pig Latin is a data flow language that is used to describe step-by-step processing of the data. |
| Support to relational/ non-relational data | Provides support to relational data | Does not support relational data | Supports both relational and non-relational data |
| Data model | Data is organized in the form of tables, partitions and buckets | Data is organized in the form of rows and columns. Rows and columns can be grouped together to form column families. | Data doesn't need to be in the form of tables. Pig is capable of taking data in any format. Doesn't require a pre-defined schema. |
| Type of queries suitable for | Suitable for large scale data analysis based queries. Unsuitable for OLTP based queries. | Suitable for analysis of large scale non-relational data. Suitable for tasks that require handling fast, random access of unstructured data. | Suitable for large scale data analysis based queries. Unsuitable for OLTP based queries. |

**d. Apache Hive**

Apache Hive [TSJS09] is an open-source data warehousing solution built on top of the MapReduce programming framework and is a part of the Hadoop software stack. Queries in Hive can be written in HiveQL which has syntax similar to SQL. The SQL-like syntax of Hive and also the support it provides to both relational and non-relational databases has made Hive a potential alternative to relational database systems.

Figure 2.3 shows a schematic diagram of the Hive architecture [TSJS09]. The main components of Hive [TSJS09] are listed below:

- **External Interfaces**, such as the Command Line Interface (CLI), web User Interface (UI) and Application Programming Interfaces (APIs) such as JDBC and ODBC.
- **Hive Thrift Server,** a simple API to execute HiveQL statements and interact with the Hive services.
- **Hive Metastore,** which acts as the system catalog.
- **Driver,** which is responsible for the life cycle management of a HiveQL statement during compilation, optimization and execution phases.
- **Compiler,** which when invoked by the driver converts the HiveQL query into a directed acyclic graph (DAG) of MapReduce jobs.
- **Execution Engine,** which executes the aforementioned MapReduce jobs in topological order. Hive currently uses Hadoop as its execution engine.

Hive Metastore is the system catalog for Hive [TSJS09], where it stores all the metadata related to the tables stored in Hive. The Metastore contains schemas and statistics that are useful for data analysis and exploration [TSJS09]. The Metastore is the reason Hive is categorized as a data warehouse solution when compared to the other data processing solutions such as Pig [PIG14]. A Metastore is stored in either a relational database such as MySQL or a file system such as NFS (Network File System) but not HDFS, since HDFS is optimized for sequential scans and not randomized scans [TSJS09].

The Metastore is made up of the following objects [TSJS09]:

- **Database:** a namespace for tables. If the user does not specify a database name, a default is used.
- **Table:** stores the metadata of the list of columns, their types, owner and storage information and also the serialization/deserialization information.
- **Partition:** stores the metadata regarding the columns, serialization/deserialization information and storage information for the partitions of the tables.

**Figure 2.3** *Hive architecture [TSJS09]*

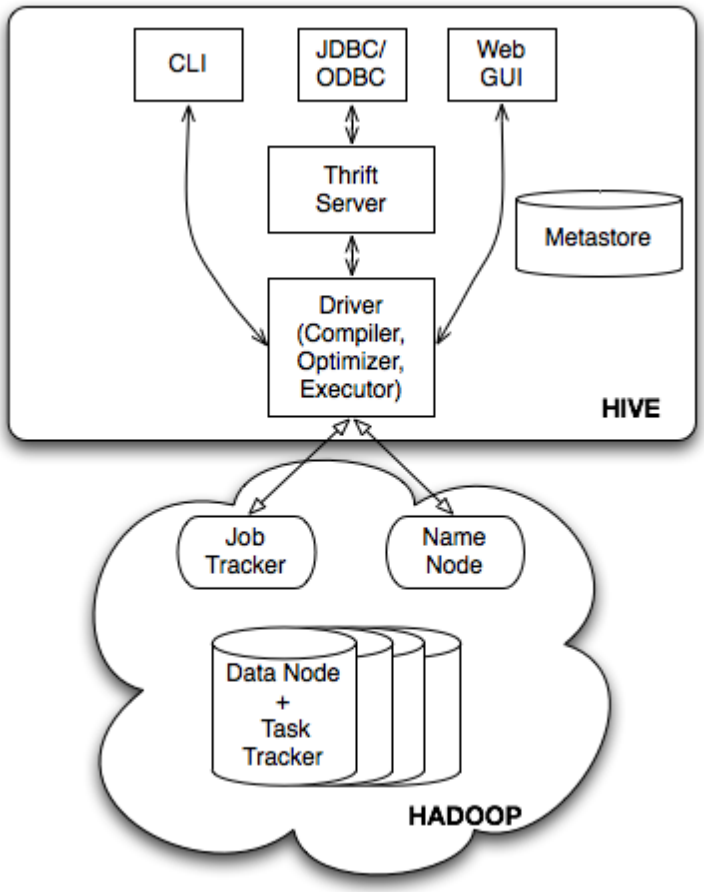
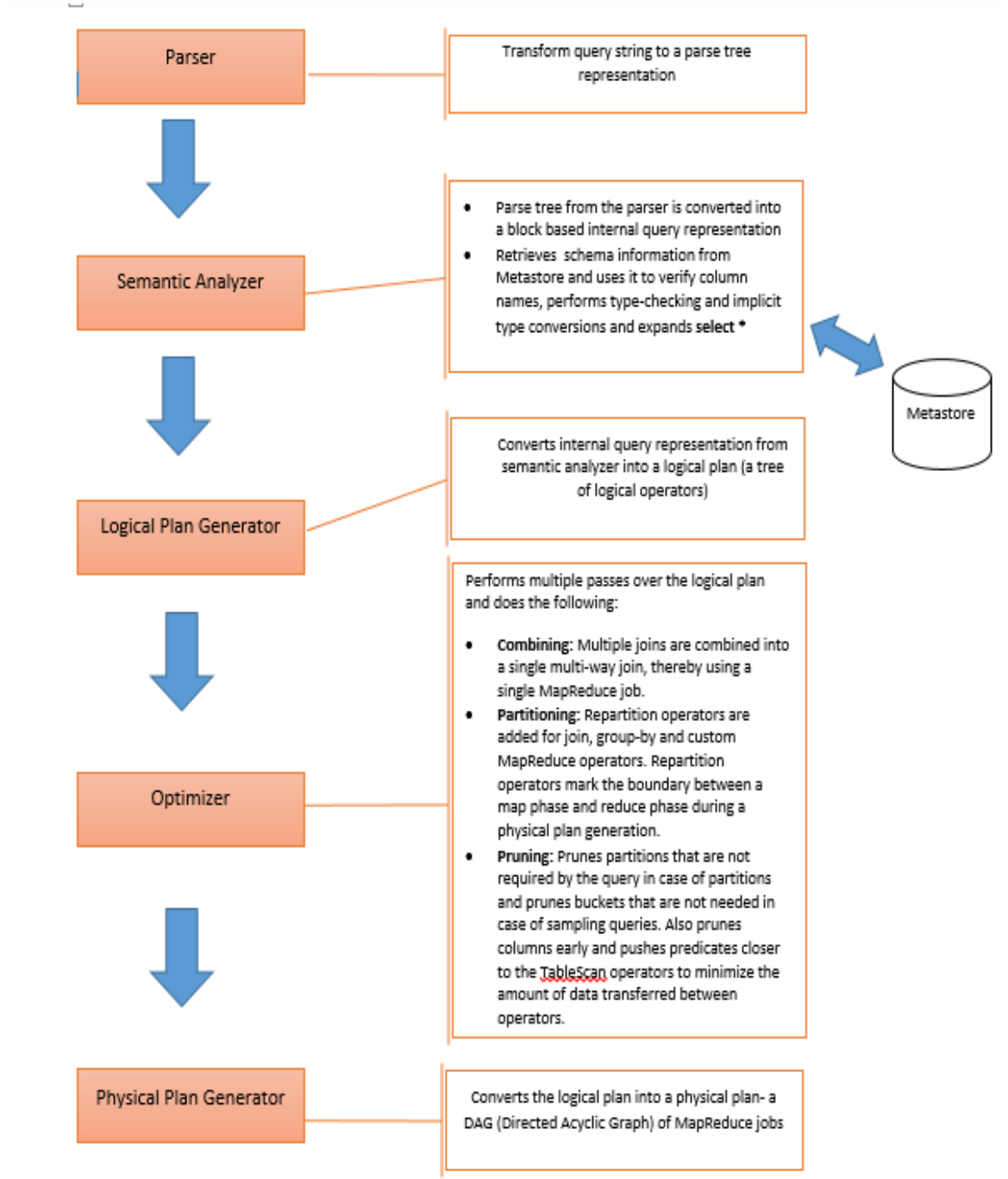



Figure 2.4 gives a schematic diagram of the order in which the compiler converts a HiveQL query into a DAG (Directed Acyclic Graph) of MapReduce tasks [TSJS09]. The driver invokes the compiler once the user executes the query. As shown in Figure 2.4, different components of the Hive compiler process the query and in the end generate a DAG of MapReduce jobs.

**Figure 2.4** *Order of execution of the Hive compiler* [*TSJS09]*




In order to illustrate how Hive translates a query into a DAG of MapReduce jobs, consider the following multi-table insert query [TSJS09] that performs a join over the tables *status_updates* and *profiles* and then stores the data regarding the counts of daily status updates per school into

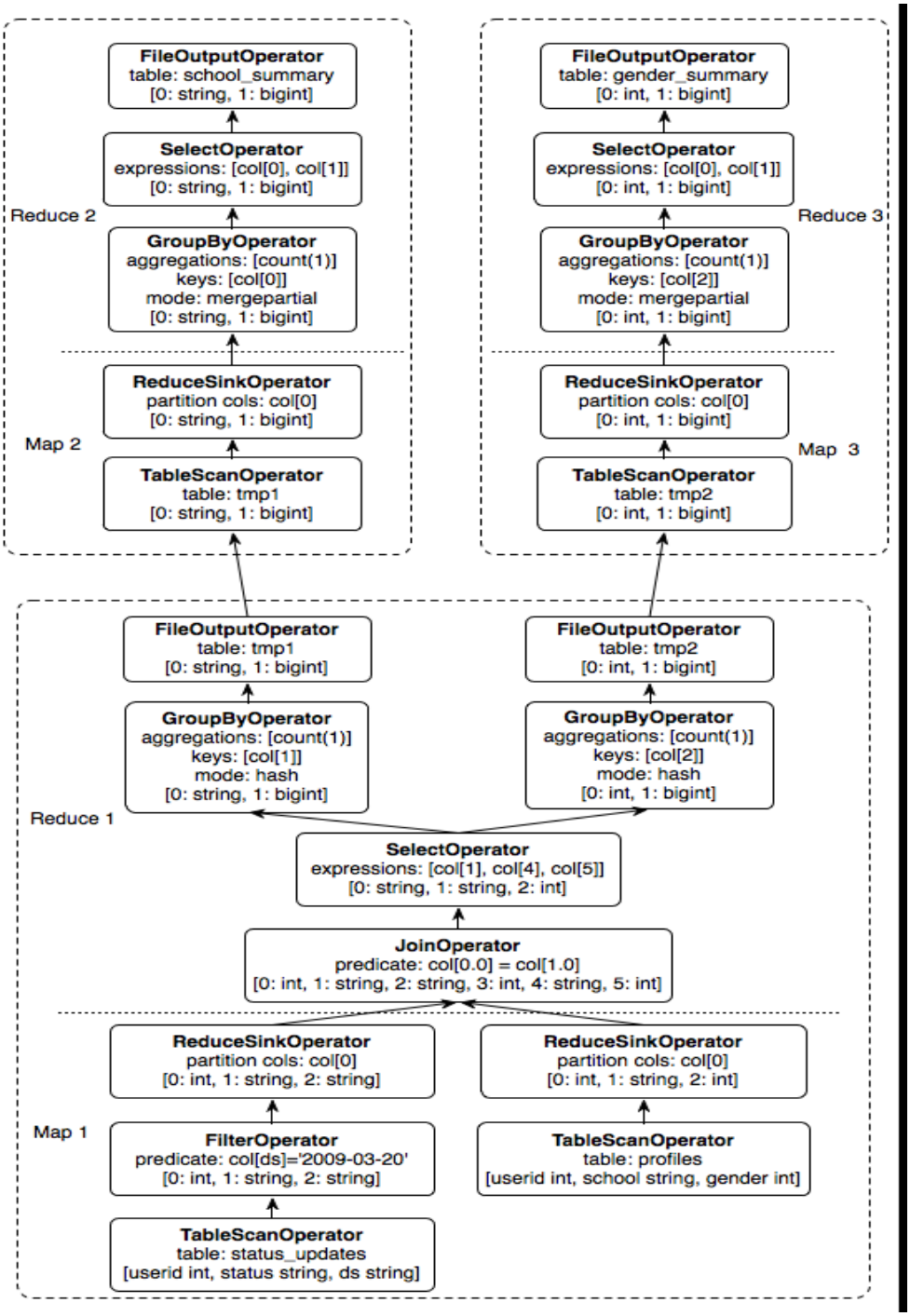
the table *school_summary* and count of daily status updates per gender into the table *gender_summary*.

**Query [TSJS09]:**

```
FROM (SELECT a.status, b.school, b.gender
    FROM status_updates a JOIN profiles b
        ON (a.userid = b.userid AND
            a.ds='2009-03-20' )) subq1
INSERT OVERWRITE TABLE gender_summary
    PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1)
GROUP BY subq1.gender
INSERT OVERWRITE TABLE school_summary
    PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1)
GROUP BY subq1.school
```

The query plan with three MapReduce jobs is as shown in Figure 2.5. The initial MapReduce job scans the two tables, *profiles* and *status_updates*. The map function filters the table *status_updates* based on the partition specified. The specified columns are then passed over to a reduce function, where a *JoinOperator* performs a join operation using the specified column (*userid*). The reduce function then runs the *groupby* aggregation specified in the two *select* sub-queries and passes the results of each of the *select* statements to a separate MapReduce job, each of which perform the similar actions to the one specified above and finally display the results. The Hive Data Model organizes data into tables, partitions, and buckets [TSJS09]. A table in Hive is similar to a table in a relational database. Each table is stored as a separate directory in the HDFS. Data stored in the table is serialized and stored as files within the directory corresponding to the table [TSJS09]. Users can specify their own serialization/deserialization formats or use the built-in formats defined in Hive [HSERDE]. The serialization format, along with the metadata about the table is stored in the Hive Metastore.

**Figure 2.5** *Query execution plan of an example Hive query [TSJS09]*




A table can have one or more than one partitions. Each new partition of a table is created as a new sub-directory within the table directory. The data corresponding to the partitions is also stored in the sub-directories accordingly. The data stored in each partition can be sub-divided into

buckets, based on the hash of a column in the table [TSJS09]. Each bucket is stored as a separate file in the directory corresponding to its partition.

**Table 2.2** *Differences between Hive and a traditional relational database [W12]*

| Feature | Hive/HiveQL | relational database/SQL |
|---|---|---|
| **Schema enforcement** | Schema on read: data is not verified at the time of loading. Data is verified when a query is executed on the table [W12] | Schema on write: if data being loaded into the table does not conform to the schema, it is rejected [W12] |
| **Updates** | Hive does not support updates or deletes. A table cannot be modified once it is created and new data can be added to it by using INSERT INTO or by creating a new partition. | Updates, delete, and insert are supported |
| **Indexing** | Supports indexing | Supports Indexing |
| **Latency** | Comparatively more latency | Very low latency |
| **Multi-table inserts** | Supported | Not supported |
| **Data types** | Integral, floating point, Boolean, string, binary, timestamp, array, map, struct | Integral, floating point, Boolean, string, temporal |
| **Supported paradigms** | Large scale analysis (Large scale OLAP) | OLTP |

While Hive supports relational schema and HiveQL follows a syntax similar to SQL, there are some differences between Hive and relational databases. Table 2.2 shows some differences between Hive/HiveQL and a relational database system/SQL such as, the difference in terms of reading data, the differences in their respective languages such as data types supported, types of updates supported by both the systems and their supported paradigms. Table 2.2 shows that Hive verifies the data only while a query is being executed on the data and does not verify the data while it is being loaded into the tables- a procedure followed by relational databases. Unlike relational databases, which support update, delete and insert operations on tables, Hive only allows data to be updated into a table by creating a new partition within or overwriting an existing table. Since Hive uses the MapReduce programming paradigm, which is a batch process, running a HiveQL query involves more latency than a SQL query. Because of its lack of support to update and delete operations and the latency involved with MapReduce, Hive is more suitable for large

scale OLAP type operations, while relational databases with their support to quick update and delete operations support OLTP operations.

**Note:** One limitation of HiveQL is that it currently only supports equi-joins [CWR12].

**e. Sqoop**

Sqoop is a data transfer tool that is designed to transfer data between Hadoop and other structured data management systems such as relational databases [SQOOP]. By using Sqoop, bulk data can be transferred efficiently between Hadoop and other data management systems. Figure 2.1 shows how Sqoop acts as an interface between Hadoop and outside relational data management systems.

## 2.2 Big Data Benchmarking

Benchmarking is the process of comparing the performance of a system against a standard reference. Several standard benchmarks such as TPC-C [TPCC], TPC-H [TPCH], and TPC-DS [TPCDS] have been developed. As the number of big data management systems is increasing every day, there is a need to compare and evaluate the performance and price of these systems. Currently, organizations are working toward an industry standard benchmark to compare and evaluate the performance of different big data management systems.

An industry standard big data benchmark should be an end-to-end benchmark, covering all the features of a big data: *volume, variety* and *velocity* [GRHR13]. Volume refers to the terabyte scale data that is typically managed by a big data management system. Variety refers to different types of data such as structured and non-structured data that is stored in a big data management system. Velocity refers to the higher data arrival rates such as click streams.

While there have been several benchmarking standards defined for evaluating the performance of the Hadoop ecosystem, such as Sorting programs (Hadoop Sort Program [HSORT], TeraSort [TSORT]), GridMix [GMIX] and HiBench Benchmarking Suite [HHDX10]), none of them have well-defined queries or schemas necessary for evaluating the run time performance of a big data management system such as Hive. In order to effectively benchmark the performance of the big

data systems, we identified benchmarks such as BigBench [GRHR13] and Hive Performance Benchmark [HPB]. Both BigBench and Hive Performance Benchmark provide a schema, queries, and data generator and also support structured and unstructured data. BigBench adopts the structural part of its schema from the TPC-DS benchmark [NP06]. Ahmad et al. improved the TPC-DS schema by adding semi-structured and unstructured components. At the time of writing this thesis, Ahmad et al. are still finalizing the implementations of their data and query generators [GRHR13].

Since TPC-DS is designed for relational data, it does not answer the *variety* requirement of an industry standard end-to-end big data benchmark. However, since TPC-DS is designed to scale to several hundred petabytes, and has a refresh process defined, it satisfies the *volume* and *velocity* requirements. Since the motive of this thesis is to observe how a big data management system such as Hive performs with large scale relational data, i.e., *volume*, we will use the TPC-DS benchmark to analyze the performance of Hive. In the following section, we provide a brief overview of the schema used by the TPC-DS benchmarking standard. Appendix A gives the procedure to use the dbgen data generator of TPC-DS. The schema and queries used by TPC-DS are described in Chapter 3.

In Chapter 2, we describe the general features of the Hadoop software stack, the MapReduce programming model and the Hive data warehouse platform. In Chapter 3, we describe the practical aspects such as the steps followed in setting up the hardware. We also describe the architecture of the Hadoop cluster used in this thesis and the hardware and software settings of the machines used to set up the Hadoop cluster used in this thesis.

## Chapter 3: Hardware and Software Settings

The Hadoop cluster that was set up for the purpose of this thesis was named the CSHadoop cluster. CSHadoop is a 4 node cluster. For setting up the CSHadoop cluster, we approached Dr. Paul Talaga, Assistant Professor-Educator at the University of Cincinnati. Dr. Talaga and his team have been managing an OpenStack Cloud Cluster at University of Cincinnati called the CSCloud. Under Dr. Talaga's guidance, we set up the CSHadoop cluster with four nodes, and a MySQL machine with the hardware configuration mentioned in Table 3.1. We equipped each of the four machines in the cluster with an additional 3.0 TB SATA hard drive in addition to the 80 GB hard drive already present in the machines to facilitate enough space for the data on the Hadoop Distributed File System (HDFS). Dr. Talaga helped us in setting up the networking between all the machines. We selected the Hortonworks HDP 1.3.2 platform [HINST] to set up a Hadoop cluster since it has a well-documented approach to setup a cluster. Hortonworks HDP 1.3.2 uses Apache Ambari [AAMB], which provides an interface that facilitates automated setup, deployment and maintenance of a Hadoop cluster. Based on the software requirements for installing the Hortonworks HDP 1.3.2 [HINST], we installed the CentOS 6.4 minimal operating system. After following the pre-deployment procedure specified in the Hortonworks HDP 1.3.2 manual [HINST], we deployed the CSHadoop cluster. We describe the architecture of the CSHadoop cluster, and how different modules of the Hadoop software stack are assigned to different machines in the cluster in section 3.3. The hardware and software configurations of all the machines used in this thesis have been explained in further detail in sections 3.1 and 3.2. We also describe the specifications of the switch used to connect all the machines together in the cluster in section 3.1. In section 3.4, we describe the features of Query 7 of TPC-DS benchmark, which we selected as a representative OLAP-style query. We also describe the schema of different tables used in Query 7 in section 3.4.

## 3.1 Hardware Configuration

### 3.1.1 Machines used

#### a. CSHadoop cluster and MySQL machine:

**Table 3.1** *CSHadoop and MySQL machine hardware configuration*

| | |
|---|---|
| **Number of Machines** | 5 |
| **Machine** | DELL-POWEREDGE-C6100-XS23-TY3 |
| **Processor** | Intel Xeon CPU 8-x-2-26GHz-L5520 |
| **Total RAM** | 48GB RAM each |
| **Hard drive** | CSHadoop - 3.08TB-SATA<br>MySQL machine - 2.05TB- SATA |

#### b. Cloudgate

**Table 3.2** *Cloudgate hardware configuration*

| | |
|---|---|
| **Number of Machines** | 1 |
| **Machine vendor** | HP |
| **Processor** | Dual Core AMD Opteron Processor 280 |
| **Total RAM** | 8GB |
| **Hard drive** | 250GB |

### 3.1.2 Network Information

- All 6 machines used in this experiment are connected together using a Cisco switch as shown below in Table 3.3.

**Table 3.3** *Network configuration for CSHadoop and MySQL machine*

| | |
|---|---|
| **Switch** | Cisco SG 200-26 26-Port Gigabit Smart Switch |

### 3.2 Software Configuration

The CSHadoop cluster was configured using Hortonworks HDP 1.3.2 [HHDP], an open-source Hadoop distribution. The version of MySQL used on the MySQL machine is 5.1.71. The versions of different software used in the cluster are listed in Table 3.4.

**Table 3.4** *Hadoop software versions installed in CSHadoop [HHDP]*

| | |
|---|---|
| **Operating System** | Centos 6.4 Minimal |
| **Hortonworks version** | 1.3.2 |
| **Hadoop version** | 1.2 |
| **Hive version** | 0.11 |
| **MySQL version** | 5.1.71 |

### 3.3 CSHadoop Architecture

A typical Hadoop cluster is made up of the following types of nodes [TNHDP]:

- Primary nodes (HDFS namenode, secondary namenode, MapReduce job tracker)
- Secondary nodes (HDFS datanodes, MapReduce task trackers)

The CSHadoop cluster has two primary nodes and four secondary nodes, with the primary nodes also acting as secondary nodes. The organization of the cluster and the different processes running on the cluster are shown in Figure 3.1. The four nodes are named Hadoop1, Hadoop2, Hadoop3, and Hadoop4. The list of different processes running on the different nodes is as shown in Table 3.5.

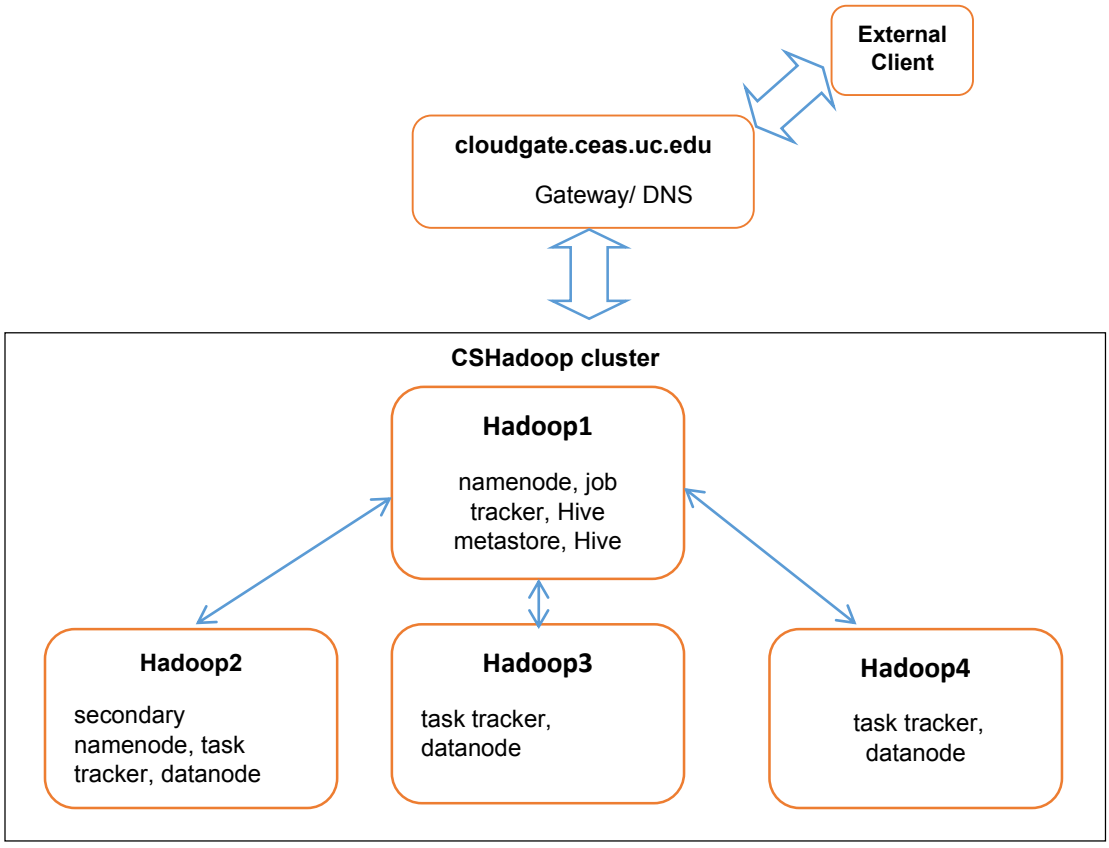
Hadoop1 acts as the namenode and stores all the details of the distributed file system (HDFS) including locations of the datanodes and locations of different files and directories stored on different datanodes. Hadoop1 also acts as the job tracker and is responsible for running and managing the MapReduce jobs across the different task tracker nodes. The Hive Metastore,

which acts as a system catalog for different tables stored on Hive runs on Hadoop1. Hive sever2, which acts as a thrift server and allows different client interfaces to connect to Hive also runs on Hadoop1. Hadoop2 acts as a secondary namenode, a back-up service that contains the same details as the namenode, and is used to manage HDFS if the namenode fails.

**Table 3.5** *CSHadoop cluster configuration*

| Node | Primary/Secondary | Process |
|---|---|---|
| **Hadoop1** | primary, secondary | namenode, job tracker, datanode, task tracker, Hive Metastore, Hive server2 |
| **Hadoop2** | primary, secondary | Secondary namenode, datanode, task tracker |
| **Hadoop3** | secondary | datanode, task tracker |
| **Hadoop4** | secondary | datanode, task tracker |

**Figure 3.1:** *CSHadoop architecture*



The four machines act as secondary nodes in the form of datanodes for data storage, and as task trackers for running the MapReduce processes. As shown in Figure 3.1, an external client accesses the CSHadoop cluster by setting up a secure connection (ssh connection) with *Cloudgate* (cloudgate.ceas.uc.edu). Cloudgate acts as the gateway for all incoming and outgoing traffic. Cloudgate also acts as the DNS server for all the nodes in CSHadoop.

### 3.4 TPC-DS Schema and Features of Query 7

The TPC-DS schema is a decision support system that models a retail product supplier [NP06]. TPC-DS follows a snowflake schema [NP06]. Our focus is on identifying how Hive performs with respect to aggregate queries. The features of Query 7 [TPCDS], selected for this thesis, are mentioned in table 3.6. Query 7 is a five table join query with four aggregation operations, one group by operation and one order by operation.

**Table 3.6** *Features of Query 7 [TPCDS]*

| Feature | Value |
|---|---|
| Number of tables | 5 |
| Number of joins | 5 |
| Number of aggregate operations | 4 |
| Number of group by operations | 1 |
| Number of order by operations | 1 |

The schemas of the five tables (*store_sales, customer_demographics, item, promotion and date_dim)* used in Query 7 are given in Table 3.7. Table 3.7 also describes the difference in data types between the SQL version of the schema definition and the HQL version of the same. Figure 3.2 shows a part of the schema of TPC-DS [NP06]. The snowflake schema shown in Figure 3.2 depicts how the fact table (*store_sales*) and some of the dimension tables (*customer_demographics, item, promotion*) that are a part of Query 7 are related to each other.

**Figure 3.2** *An excerpt of TPC-DS snowflake schema [NP06]*

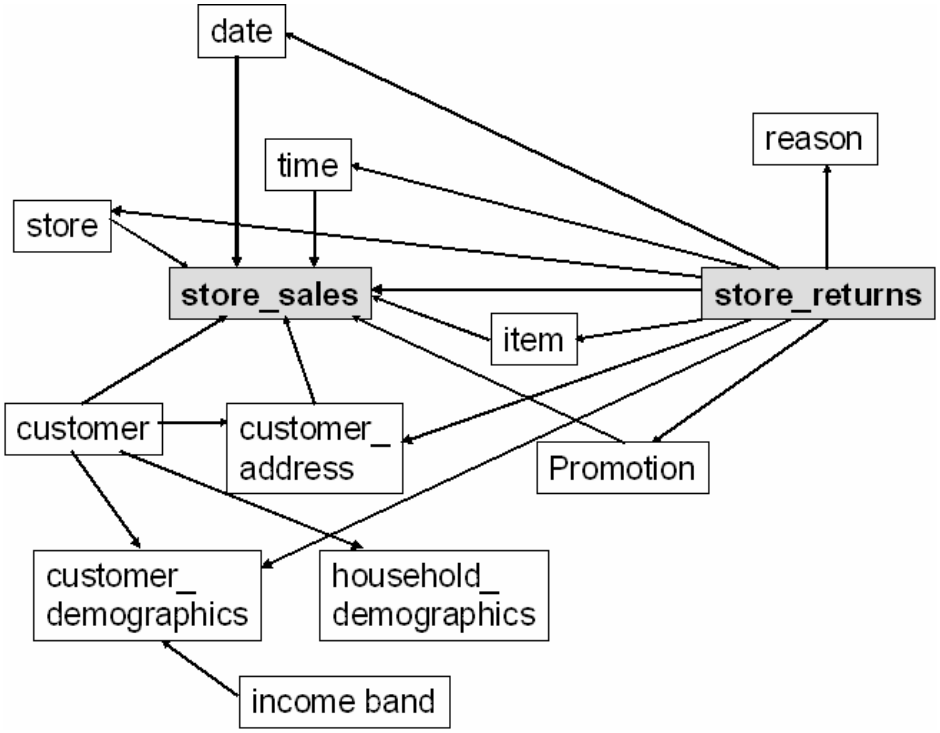



**Table 3.7** *Schema of different tables used in Query 7 [TPCDS]*

| Table name | Column Name | SQL data type definition | HiveQL data type definition |
|---|---|---|---|
| customer_demographics | cd_demo_sk | integer | int |
| | cd_gender | char(1) | string |
| | cd_marital_status | char(1) | String |
| | cd_education_status | char(20) | string |
| | cd_purchase_estimate | integer | int |
| | cd_credit_rating | char(10) | string |
| | cd_dep_count | integer | int |
| | cd_dep_employed_count | integer | int |
| | cd_dep_college_count | integer | int |
| Item | i_item_sk | integer | int |
| | i_item_id | char(16) | string |
| | i_rec_start_timestamp | date | timestamp |
| | i_rec_end_timestamp | date | timestamp |
| | i_item_desc | varchar(200) | string |
| | i_current_price | decimal(7,2) | decimal |
| | i_wholesale_cost | decimal(7,2) | decimal |
| | i_brand_id | integer | int |
| | i_brand | char(50) | string |
| | i_class_id | integer | int |
| | i_class | char(50) | string |

| | | | |
|---|---|---|---|
| | i_category_id | integer | int |
| | i_category | char(50) | string |
| | i_manufact_id | integer | int |
| | i_manufact | char(50) | string |
| | i_size | char(20) | string |
| | i_formulation | char(20) | string |
| | i_color | char(20) | string |
| | i_units | char(10) | string |
| | i_container | char(10) | string |
| | i_manager_id | integer | int |
| | i_product_name | char(50) | string |
| Promotion | p_promo_sk | integer | int |
| | p_promo_id | char(16) | string |
| | p_start_date_sk | integer | int |
| | p_end_date_sk | integer | int |
| | p_item_sk | integer | int |
| | p_cost | decimal(15,2) | decimal |
| | p_response_target | integer | int |
| | p_promo_name | char(50) | string |
| | p_channel_dmail | char(1) | string |
| | p_channel_email | char(1) | string |
| | p_channel_catalog | char(1) | string |
| | p_channel_tv | char(1) | string |
| | p_channel_radio | char(1) | string |
| | p_channel_press | char(1) | string |
| | p_channel_event | char(1) | string |
| | p_channel_demo | char(1) | string |
| | p_channel_details | varchar(100) | string |
| | p_purpose | char(15) | string |
| | p_discount_active | char(1) | string |
| store_sales | ss_sold_date_sk | integer | int |
| | ss_sold_time_sk | integer | int |
| | ss_item_sk | integer | int |
| | ss_customer_sk | integer | int |
| | ss_cdemo_sk | integer | int |
| | ss_hdemo_sk | integer | int |
| | ss_addr_sk | integer | int |
| | ss_store_sk | integer | int |
| | ss_promo_sk | integer | int |
| | ss_ticket_number | integer | int |
| | ss_quantity | integer | int |
| | ss_wholesale_cost | decimal(7,2) | decimal |
| | ss_list_price | decimal(7,2) | decimal |
| | ss_sales_price | decimal(7,2) | decimal |
| | ss_ext_discount_amt | decimal(7,2) | decimal |
| | ss_ext_sales_price | decimal(7,2) | decimal |
| | ss_ext_wholesale_cost | decimal(7,2) | decimal |
| | ss_ext_list_price | decimal(7,2) | decimal |
| | ss_ext_tax | decimal(7,2) | decimal |
| | ss_coupon_amt | decimal(7,2) | decimal |

| | | | |
|---|---|---|---|
| | ss_net_paid | decimal(7,2) | decimal |
| | ss_net_paid_inc_tax | decimal(7,2) | decimal |
| | ss_net_profit | decimal(7,2) | decimal |
| date_dim | d_date_sk | integer | int |
| | d_date_id | char(16) | string |
| | d_date | date | timestamp |
| | d_month_seq | integer | int |
| | d_week_seq | integer | int |
| | d_quarter_seq | integer | int |
| | d_year | integer | int |
| | d_dow | integer | int |
| | d_moy | integer | int |
| | d_dom | integer | int |
| | d_qoy | integer | int |
| | d_fy_year | integer | int |
| | d_fy_quarter_seq | integer | int |
| | d_fy_week_seq | integer | int |
| | d_day_name | char(9) | string |
| | d_quarter_name | char(6) | string |
| | d_holiday | char(1) | string |
| | d_weekend | char(1) | string |
| | d_following_holiday | char(1) | string |
| | d_first_dom | integer | int |
| | d_last_dom | integer | int |
| | d_same_day_ly | integer | int |
| | d_same_day_lq | integer | int |
| | d_current_day | char(1) | string |
| | d_current_week | char(1) | string |
| | d_current_month | char(1) | string |
| | d_current_quarter | char(1) | string |
| | d_current_year | char(1) | string |

The experimental procedure, results, analysis, and conclusions are described in Chapter 4.

# Chapter 4: Experimental Setup and Results

In Chapter 2, we describe the features of the Hadoop software stack, MapReduce parallel programming framework, and Hive, the parallel data warehouse that runs on top of the MapReduce framework. We also identify and describe the features of the TPC-DS benchmark. Chapter 3 describes the architecture of the Hadoop cluster set up for the purpose of this thesis and the features of Query 7 used in the experimental procedure described in Chapter 4. In order to compare the performance of Hive with a relational database, we use a MySQL database described in Chapter 3. In this chapter, we describe the experimental procedure for the thesis and the results obtained from the experiments. We analyze the results obtained and define some conclusions. We also describe the issues we encountered while using the cluster.

## 4.1 Experimental Procedure

The main goal of this experimental procedure is to analyze how Hive performs in comparison to a relational database in terms of table definition, data loading, and query execution. We chose Query 7 of TPC-DS, which is an OLAP-style representative query. In order to compare both the systems, we generate datasets of eight different sizes, ranging between 7.5 GB to 390 GB and analyze the results obtained by running Query 7 on all these datasets. The following steps describe the experimental procedure.

**Step 1** *Generating data and queries*

Appendix A describes the procedure to download and run the necessary tools to generate the queries and data required for the experiment. Since Query 7 uses only a subset of the entire TPC-DS dataset, we use the procedure described in Appendix A to generate the entire dataset, and use only the subset necessary for Query 7.

**Step 2** *Defining the tables specified in Query 7*

The SQL code for defining the table *item* [TPCDS] and its corresponding HiveQL implementation is shown in Table 4.1. Table 4.1 depicts the difference in the data types used in the SQL implementation of the table definition and the Hive version of the same. Besides the fundamental difference in naming the data types such as integer (*integer* in MySQL and *int* in Hive), other differences such as defining a date type data type can also be seen. While MySQL has a *date* data type defined, Hive does not support a date data type in the version used in this thesis (0.11). All data that is to be represented in the form of a date is specified as a *TIMESTAMP* in Hive. In Hive 0.11, all character data is represented as a *string* while they can be separated as *char* and *varchar* in SQL. Tables 4.5.2-4.12.2 show the amount of time taken for defining the schema for the datasets in both MySQL and Hive.

**Table 4.1** *item table definition in SQL and HiveQL [TPCDS]*

| **SQL implementation** | **HiveQL implementation** |
|---|---|
| create table item<br>(<br>i_item_sk integer not null,<br>i_item_id char(16) not null,<br>i_rec_start_date date,<br>i_rec_end_date date,<br>i_item_desc varchar(200),<br>i_current_price decimal(7,2) ,<br>i_wholesale_cost decimal(7,2),<br>i_brand_id integer,<br>i_brand char(50),<br>i_class_id integer,<br>i_class char(50),<br>i_category_id integer,<br>i_category char(50),<br>i_manufact_id integer,<br>i_manufact char(50),<br>i_size char(20),<br>i_formulation char(20),<br>i_color char(20),<br>i_units char(10),<br>i_container char(10),<br>i_manager_id integer,<br>i_product_name char(50),<br>primary key (i_item_sk)<br>); | create table item<br>(<br>i_item_sk int,<br>i_item_id string,<br>i_rec_start_timestamp timestamp,<br>i_rec_end_timestamp timestamp,<br>i_item_desc string,<br>i_current_price decimal,<br>i_wholesale_cost decimal,<br>i_brand_id int,<br>i_brand string,<br>i_class_id int,<br>i_class string,<br>i_category_id int,<br>i_category string,<br>i_manufact_id int,<br>i_manufact string,<br>i_size string,<br>i_formulation string,<br>i_color string,<br>i_units string,<br>i_container string,<br>i_manager_id int,<br>i_product_name string<br>)<br>row format delimited fields terminated by '\|'<br>lines terminated by '\n'; |

**Step 3** *Loading the data into MySQL and Hive*

The next step is to load the data that was previously generated in Step 1 into the tables defined in Step 2. Table 4.2 describes the SQL and HiveQL commands to load the data for the *store_sales* table into MySQL and Hive respectively. Table 4.2 shows that the SQL and HiveQL commands have a similar syntax to load the data into their respective systems.

**Table 4.2** *SQL and HiveQL commands to load the data for store_sales table into MySQL and Hive respectively*

| SQL implementation | HiveQL implementation |
|---|---|
| load data infile<br>'/karthikTemp/datasets/100gb/store_sales.dat'<br>replace into table store_sales<br>fields terminated by '\|' lines terminated by '\n'; | load data local inpath<br>'/data/datasets/dsgen/tools/100gb/store_sales.dat'<br>overwrite into table store_sales; |

**Step 4** *Execute the query*

Once the data is loaded into the system, the next step is to execute the query in both the systems and collect the results. The original SQL version of the query is shown in Figure 4.1.

**Figure 4.1** *Original SQL version of Query 7 [TPCDS]*

```
select top 100 i_item_id,
      avg(ss_quantity) agg1,
      avg(ss_list_price) agg2,
      avg(ss_coupon_amt) agg3,
      avg(ss_sales_price) agg4
from store_sales, customer_demographics, date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
     ss_item_sk = i_item_sk and
     ss_cdemo_sk = cd_demo_sk and
     ss_promo_sk = p_promo_sk and
     cd_gender = 'F' and
     cd_marital_status = 'D' and
     cd_education_status = 'College' and
     (p_channel_email = 'N' or p_channel_event = 'N') and
     d_year = 2001
group by i_item_id
order by i_item_id;
```

The query was modified slightly by omitting the '*top 100*' statement from *select* to get the query shown in Figure 4.2.

**Figure 4.2** *Modified SQL version of Query 7*

```
select i_item_id,
       avg(ss_quantity) agg1,
       avg(ss_list_price) agg2,
       avg(ss_coupon_amt) agg3,
       avg(ss_sales_price) agg4
 from store_sales, customer_demographics, date_dim, item, promotion
 where ss_sold_date_sk = d_date_sk and
      ss_item_sk = i_item_sk and
      ss_cdemo_sk = cd_demo_sk and
      ss_promo_sk = p_promo_sk and
      cd_gender = 'F' and
      cd_marital_status = 'D' and
      cd_education_status = 'College' and
      (p_channel_email = 'N' or p_channel_event = 'N') and
      d_year = 2001
 group by i_item_id
 order by i_item_id;
```

Since HiveQL uses joins rather than listing tables using the ',' operator in the FROM clause, the following changes were made:

**Figure 4.3** *Differences between original and HiveQL versions of Query 7*

*Original:*

```
from store_sales, customer_demographics, date_dim, item, promotion
```

*HiveQL modification:*

```
from store_sales ss join date_dim d on (ss.ss_sold_date_sk = d.d_date_sk)
join item i on ( ss.ss_item_sk = i.i_item_sk )
join promotion p on (ss.ss_promo_sk = p.p_promo_sk)
join customer_demographics cd on (ss.ss_cdemo_sk = cd.cd_demo_sk)
```

Figure 4.3 shows that each join is specified with a conditional statement that was specified in the *where* clause in the original query. The HiveQL version of the modified SQL query is as shown in Figure 4.4.

**Figure 4.4** *HiveQL version of modified SQL query from Figure 4.2*

```
select i_item_id,
       avg(ss_quantity) agg1,
       avg(ss_list_price) agg2,
       avg(ss_coupon_amt) agg3,
       avg(ss_sales_price) agg4
 from store_sales ss join date_dim d on (ss.ss_sold_date_sk = d.d_date_sk)
 join item i on ( ss.ss_item_sk = i.i_item_sk )
 join promotion p on (ss.ss_promo_sk = p.p_promo_sk)
```

```
join customer_demographics cd on (ss.ss_cdemo_sk = cd.cd_demo_sk)
    where
    cd_gender = 'F' and
    cd_marital_status = 'D' and
    cd_education_status = 'College' and
    (p_channel_email = 'N' or p_channel_event = 'N') and
    d_year = 2001
group by i_item_id
order by i_item_id;
```

The SQL and HiveQL queries mentioned in Figure 4.2 and Figure 4.4 were executed on MySQL and Hive respectively for datasets of different sizes.

**Step 5** *Repeat Steps 1-4 for different dataset sizes*

Generate datasets of different sizes ranging between 20 GB to 1TB using the procedure specified in Step1. Since the datasets used in Query 7 form a subset of the entire dataset generated, the respective sizes of the datasets used during different iterations ranges between 7.5 GB to 390 GB. Follow the procedure defined in Steps 2, 3, and 4 to define the tables, load data and run queries. Collect and analyze the execution times for each of the steps for all the dataset sizes.

**Note:** Since the dataset for the TPC-DS benchmark scales in terms of discrete scale factors [NP06] such as 100 GB, 300 GB, 1000 GB, 3000 GB, and TPC-DS benchmark specifies that datasets of other scale are considered invalid [NP06], we specify the results of the three recommended dataset sizes in the next section, while we specify the results obtained by running Query 7 on the datasets of all the other sizes in Appendix B.

The following section contains tables which display the amount of time taken to execute the previous steps on a 39 GB, a 117 GB, and a 390 GB dataset. In the following sections, Tables 4.3.1, 4.4.1, and 4.5.1 contain the total size and number of records contained in each dataset. Tables 4.3.2, 4.4.2, and 4.5.2 contain the amount of time to execute the queries specified in this section.

### 4.2 Results

In this section, we describe the results of the experimental procedure specified in previous section. We specify the amount of time taken to define the tables, load data into tables, and execute Query 7.

1. **Dataset 1: Total data set size: 100 GB**

   **Total data set size for the query: 39 GB**

**Table 4.3.1** *Total size and number of records in datasets used for the query–Dataset 1*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 56 MB | 204,000 |
| Promotion | 123 KB | 1,000 |
| store_sales | 39 GB | 287,997,024 |
| date_dim | 9.9 MB | 73,049 |

**Table 4.3.2** *Table definition, data loading and query execution times–Dataset 1*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.25s | 1.439s | 5.78s | 2.764s | 8m49.18s | 4m12.816s |
| Item | 0.11s | 0.188s | 1.73s | 2.045s | | |
| Promotion | 0.10s | 0.16s | <0.01s | 0.41s | | |
| store_sales | 0.10s | 0.204s | 14h25m47.22s | 21m43.907s | | |
| date_dim | 0.10s | 0.175s | 0.5s | 0.6s | | |

2. **Dataset 2: Total data set size: 300 GB**

   **Total data set size for the query: 117 GB**

**Table 4.4.1** *Total size and number of records in datasets used for the query–Dataset 2*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 72 MB | 264,000 |
| Promotion | 159 KB | 1,300 |
| store_sales | 116 GB | 864,001,869 |
| date_dim | 10 MB | 73,049 |

**Table 4.4.2** *Table definition, data loading and query execution times–Dataset 2*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.14s | 0.366s | 5.82s | 2.507s | 31m59.77s | 11m57.084s |
| Item | 0.11s | 0.132s | 2.24s | 2.271s | | |
| Promotion | 0.12s | 0.143s | 0.03s | 0.407s | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| store_sales | 0.11s | 0.168s | 1d19h42m48.84s | 1h2m36.776s | | |
| date_dim | 0.11s | 0.127s | 0.37s | 0.718s | | |

3. **Dataset 3: Total data set size: 1 TB**

   **Total data set size for the query: 390 GB**

   **Table 4.5.1** *Total size and number of records in datasets used for the query–Dataset 3*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 82 MB | 300,000 |
| Promotion | 184 KB | 1,500 |
| store_sales | 390 GB | 2,879,987,999 |
| date_dim | 10 MB | 73,049 |

**Table 4.5.2** *Table definition, data loading and query execution times–Dataset 3*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.12s | 0.516s | 5.74s | 3.28s | 1h35m46.23s | 38m0.41s |
| Item | 0.11s | 0.593s | 2.53s | 2.168s | | |
| Promotion | 0.09s | 0.176s | 0.02s | 0.458s | | |
| store_sales | 0.10s | 0.184s | 6d 3h17m20.76s | 2h58m8.888s | | |
| date_dim | 0.10s | 0.385s | 0.36s | 0.686s | | |

## 4.3 Observations

### a. Table definition

i. From Tables 4.3.2, 4.4.2, and 4.5.2, it can be observed that the maximum amount of time taken for defining a schema is 0.25s for MySQL and 1.439s for Hive. The minimum time taken is 0.176s for MySQL and 0.132s for Hive.

ii. The schema for the tables was defined in the following order: *customer_demographics, item, promotion, store_sales, date_dim.* It can be observed that in all the three iterations specified above, the first table that is defined (*customer_demographics*) takes longer in Hive than the other tables.

### b. Data load

We compare the amount of time taken to load datasets of different sizes. Consider the amount of time taken to load the *promotion, item and store_sales* datasets from Tables

4.3.2, 4.4.2, and 4.5.2. While the *promotion* dataset is a very small dataset, with its size ranging between 123 KB-184 KB, the *item* dataset is much larger, ranging between 56 MB-82 MB. The *store_sales* dataset is the largest dataset used in the query, ranging between 39 GB–390 GB.

*i.* *promotion dataset*: Tables 4.3.2, 4.4.2 and 4.5.2 show that while MySQL takes around 0.01–0.03s to load the datasets, Hive does the same in 0.41–0.458s.

*ii.* *item dataset:* Tables 4.3.2, 4.4.2 and 4.5.2 show that while MySQL takes between 1.73–2.53s to load the different item datasets, Hive takes between 2.045–2.168s to do the same.

*iii.* *store_sales dataset*: Tables 4.3.2, 4.4.2, and 4.5.2 show that MySQL takes over 14 hours to load a 39 GB dataset, over 1 day 19 hours to load a 116 GB dataset and over 6 days to load a 390 GB dataset. On the other hand, Hive takes over 21 minutes to load the 39 GB dataset, around 1 hour to load the 116 GB dataset and approximately 3 hours to load a 390 GB dataset.

**c. Query execution:**

i. Tables 4.3.2, 4.4.2 and 4.5.2 show that while MySQL executes the Query 7 on a 39 GB dataset in approximately 8 minutes, Hive does the same in around 4 minutes. When the same query is executed on a 117 GB dataset, MySQL takes nearly 40 minutes to execute the query, while Hive was able to perform the same query in nearly 12 minutes. On a 390 GB dataset, while Hive executed the query in 38 minutes, MySQL took nearly 1 hour 36 minutes to execute the query.

ii. Scalability: Tables 4.3.2, 4.4.2 and 4.5.2 also show that as the size of the datasets has increased from 39 GB to 390 GB, the amount of time taken by MySQL to execute the query raised from 8 minutes to 1 hour 36 minutes, an increase of nearly 16 times for a dataset size increase of 10 times. For the same datasets, the amount of time taken by

Hive increases from 4 minutes to nearly 40 minutes, which is an increase of nearly 10 times.

**d. Issues encountered while working on Hive:**

While working on the thesis, we observed that when the MySQL service which was running as the Metastore for Hive crashed, Hive would no longer run, since it would not be able to access the metadata regarding its databases and tables.

### 4.4 Discussion

From the observations made in the previous section, the following conclusions can be drawn:

**a. Data definition:**

We conclude that MySQL is marginally faster than Hive when defining the schema. However, this difference is only a fraction of a second.

**b. Data load:**

From the observations made regarding the amount of time taken to load the data for the *promotion, item,* and *store_sales* datasets, the following conclusions can be drawn:

i. *promotion* dataset: Figure 4.5 displays a comparison of the amount of time taken to load the *promotion* dataset in both MySQL and Hive. Figure 4.5 shows that MySQL is marginally faster than Hive when loading small datasets.

ii. *item* dataset: Figure 4.6 displays a comparison of the amount of time taken to load the *item* dataset in both MySQL and Hive. Figure 4.6 shows that MySQL is marginally faster than Hive when the dataset size is small. However, Figure 4.6 also shows that as the size of the *item* dataset increases, the difference in time taken to load the dataset between MySQL and Hive decreases. For the largest dataset size that was used, Hive is faster than MySQL.

iii. *store_sales* dataset: Figure 4.7 displays a comparison of the amount of time taken to load the *store_sales* dataset into both MySQL and Hive. Figure 4.7 shows that Hive is significantly faster than MySQL when loading gigabyte scale data into the system.

iv. The conclusions drawn above can be explained by the fact that when data is loaded into Hive from an external file system, Hive copies the file verbatim and does not attempt to parse the file [W12]. When the table is created, a folder with the table name is stored under the folder /user/hive/warehouse/ on HDFS or the location specified by the user under the *hive.metastore.warehouse.dir* property. When the load command is used together with the overwrite condition, the original table folder is deleted and a new folder is created and the files related to the dataset are copied into this folder. On the other hand, MySQL parses the data while loading it into the system. As a result, as the dataset size increases, MySQL takes longer to load the data into the system than Hive. For very large datasets, MySQL takes significantly longer.

**c. Query execution:**

i. Figure 4.8 shows a comparison of the execution times of both Hive and MySQL for all the datasets shown in the previous section and in Appendix B. Figure 4.8 shows that for all the dataset sizes used in this experiment, Hive was faster than MySQL while executing Query 7.

ii. Scalability: Figure 4.9 shows that Hive scales much better than MySQL. Figure 4.9 also shows that for all the datasets we have used in this experiment, Hive has linear scalability.

iii. As explained in Chapter 2, whenever a query is executed on Hive, it is split into a series of MapReduce jobs, each of which act on the data in parallel and then consolidate the results into a single section. Since the entire computing process is being split across

several machines and being executed in parallel, we can observe that Hive is faster than MySQL while executing the query.

**d. Issues encountered while working on Hive:**

i. From the observations specified in the previous section, it can be concluded that Hive is dependent on the Metastore and that the Metastore can be categorized as a single point of failure.

**Figure 4.5** *A comparison of data load times for promotion dataset between MySQL and Hive*

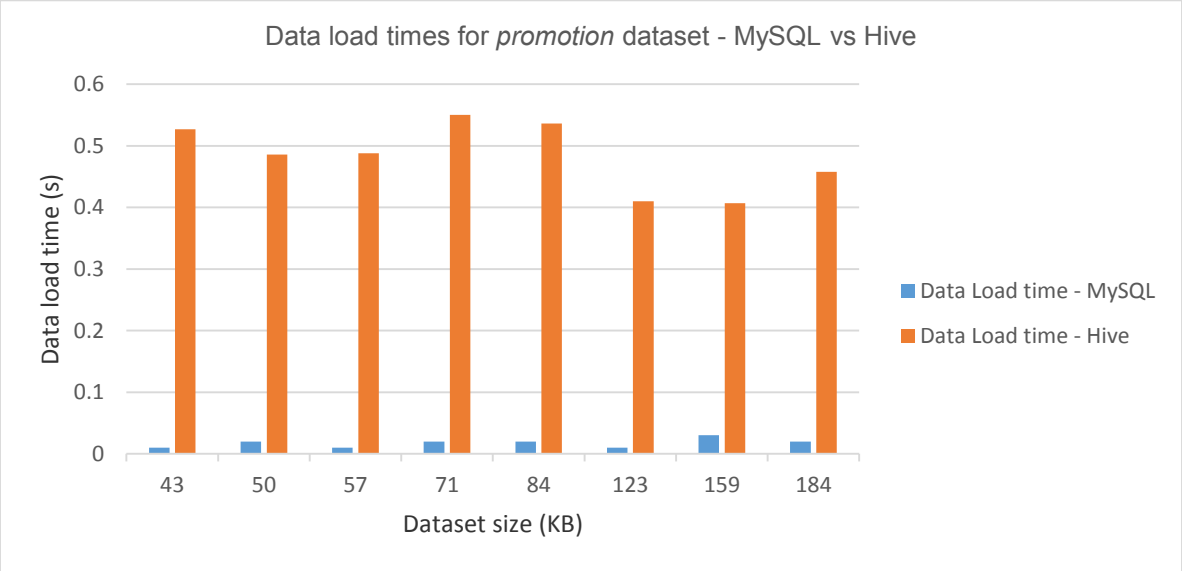



**Figure 4.6** *A comparison of data load times for item dataset between MySQL and Hive*

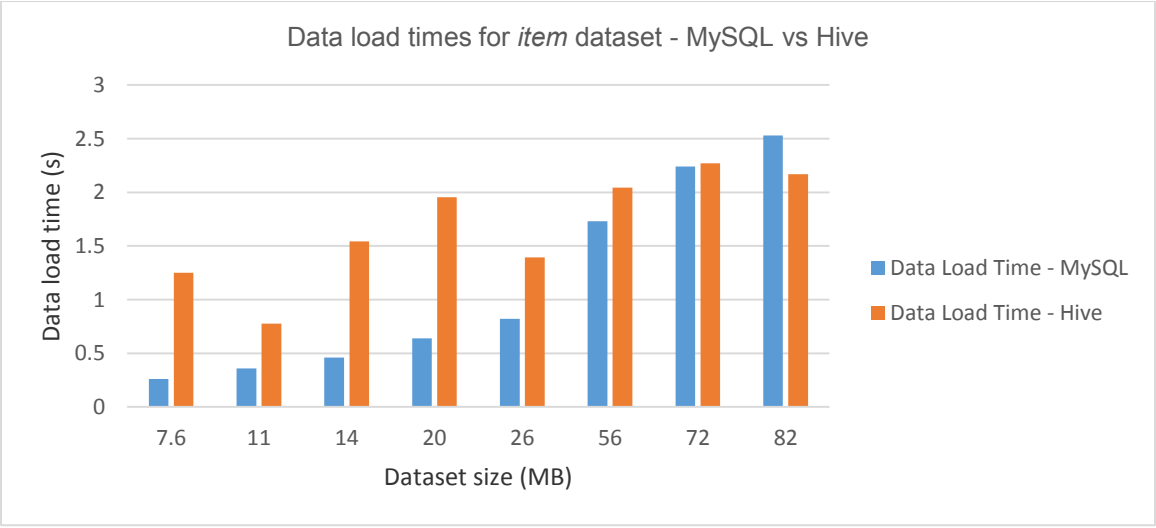


**Figure 4.7** *A comparison of data load times for store_sales dataset between MySQL and Hive*

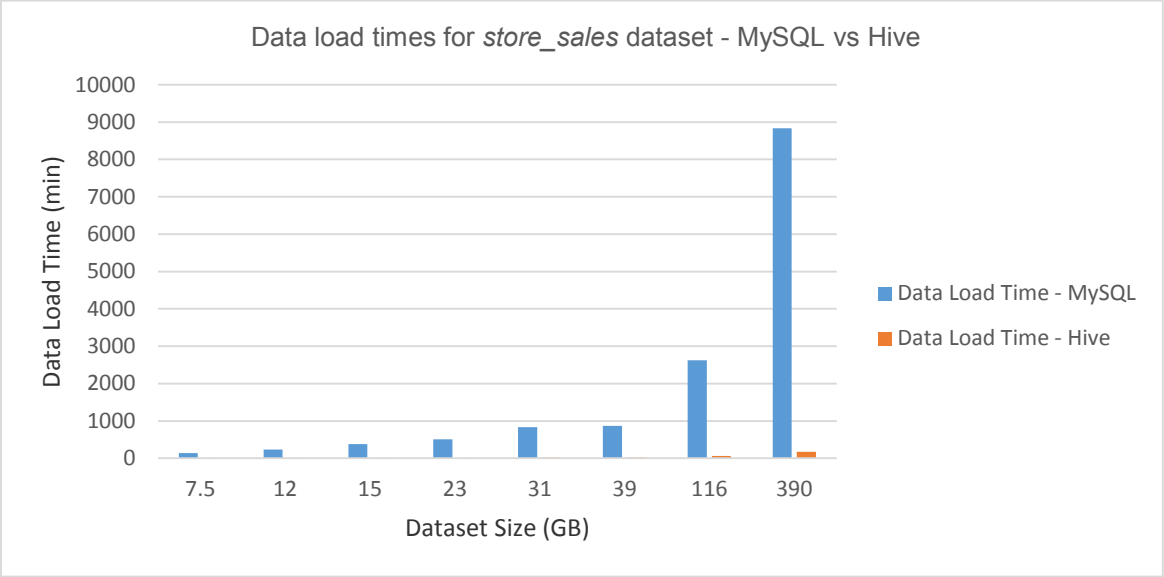



**Figure 4.8** *A comparison of query execution time for Query 7 between MySQL and Hive*

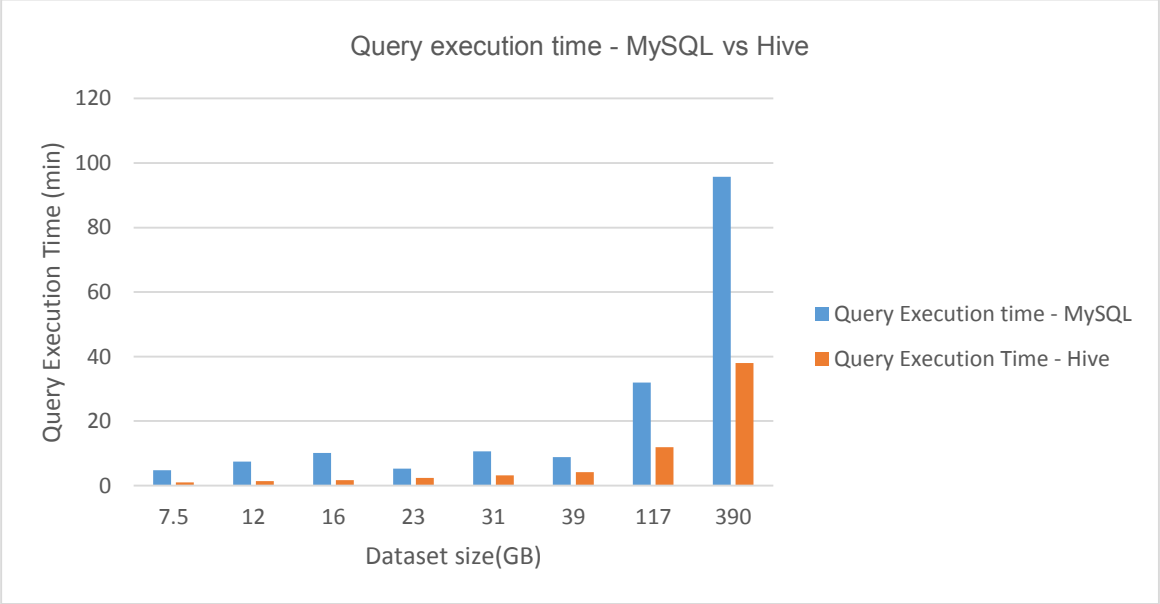
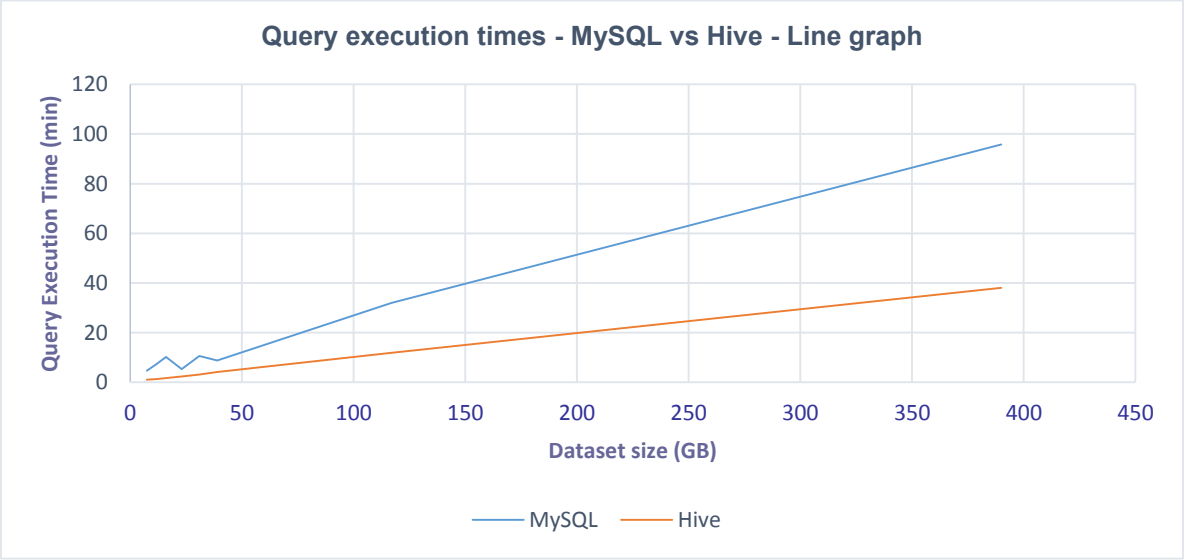


**Figure 4.9** *Scalability comparison between MySQL and Hive for executing Query 7*




There have been other studies which have benchmarked the performance of Hive since 2009. The Hortonworks Stinger Initiative [HSTNGR13] is an initiative started by Hortonworks to improve the performance of Hive used Queries 27 and 95 from the TPC-DS benchmark. Their focus was on analyzing query performance for different versions of Hive such as Hive 0.11, which is the version of Hive used for this thesis, Hive 0.12 and Hive 0.13, which are the later version to the one used in this thesis. The focus of this thesis was on analyzing the behavior of both data load and query execution times for a small Hadoop cluster. Shi et al. [SMZH10] and the Hive Performance Benchmark [HPB] both use the data and queries defined by Pavlo et al. [PPRA09] to benchmark older versions of Hive. However, their dataset sizes are limited to about 110 GB. Shi et al. benchmark query execution and data load times for Hive 0.6 using simple select, range and single aggregation queries on a 20-node cluster. They do not focus on complex queries using large number of aggregations and joins as this thesis does. The Hive Performance Benchmark defines a join query in addition to the queries used by Shi et al. It benchmarks Hive trunk version 786346, which is older than the version used in this thesis (0.11) and as explained earlier, focuses on a single dataset which is about 110 GB in size. It also benchmarks only query execution time

and not data load time. The project conducted by Pansare et al. [PZ10] is focused on identifying if Hive is suitable for mid-level data analysis. Pansare et al used the TPC-H benchmark to analyze the performance of Hive on a 4-node cluster in 2010. In addition to data load times, Pansare et al. focused on analyzing Hive's performance for different TPC-H queries while varying the number of nodes in the cluster and analyzing the processor usage by the system. However, their dataset size is limited to 10 GB, while this thesis analyzes datasets to about 400 GB. All the studies mentioned above do not compare the performance of Hive and a relational database. Study done by Jia [RPTCH] uses the TPC-H benchmark queries to benchmark Hive trunk version 799148, which is an older version of Hive on a single 100 GB dataset and compares its performance to the IBM DB2 database, while we compare Hive with a MySQL database. Jia's work does not focus on data load times. While all the studies mentioned above focus on query execution times and some such as Pansare et al. and Shi et al. focus on data load times, one additional finding of this thesis is that the Hive Metastore could become a single point of failure. This thesis also uses larger datasets and different benchmark standards than the studies mentioned above. Table 4.6 summarizes the aforementioned differences between this thesis and the other studies an.

**Table 4.6** *Comparison of current study with related studies*

| Study | Benchmark | Dataset size | Hive version | Number of nodes used | Additional Differences |
|---|---|---|---|---|---|
| Hortonworks Stinger Initiative [HSTNGR13] | TPC-DS (Query 27, 95) | 200GB, 1 TB | 0.11, 0.12, 0.13 | Not specified | • Not compared to a relational database<br>• Does not consider data load time |
| Shi et al. [SMZH10] | Queries and datasets provided by Pavlo et al. [PPRA09] | 110 GB | 0.6 | 20 | • Not compared to a relational database<br>• Uses queries that are not as complex as the one used in this study |
| Hive Performance Benchmark [HPB] | Queries and datasets provided by Pavlo et al. [PPRA09] | 110 GB | Trunk version 786346 | 11 | • Not compared to a relational database<br>• Uses queries that are not as complex as the one used in this study |

| | | | | | |
|---|---|---|---|---|---|
| Jia et al. [RPTCH] | TPC-H | 100 GB | Trunk version 799148 | 11 | • Does not consider data load times |
| Pansare et al. [PZ10] | TPC-H | 10 GB | Not specified | 4 | • Focuses on mid-level data analysis and uses only 10GB dataset<br>• Not compared to a relational database |
| Current study | TPC-DS | 390 GB | 0.11 | 4 | • Compares performance with a relational database<br>• Considers both data load time and query execution time<br>• Focuses on large scale data analysis |

### 4.5 Conclusion

In summary, we can conclude that for all the large datasets used in this experiment, Hive was able to load data faster than MySQL. For the smaller datasets, MySQL was marginally faster than Hive. When executing a five-table join query with four aggregation operations, one group by operation and one order by operation, for dataset sizes ranging between 7.5 GB to 390 GB, Hive performed consistently faster than MySQL. We also identified that the Hive Metastore is a potential single point of failure.

In the next chapter, we describe the contributions this thesis has made to research and describe different ways by which Hive can be improved in the future. We also describe the direction in which the work done in this thesis can be extended.

# Chapter 5: Contributions and Future Work

In this chapter, we give a summary of the contributions made to research and also suggest possible future work.

## 5.1 Contributions

In Chapter 2, we provide an overview of the features of the Hadoop software stack and explain the different parts of the stack that are relevant to this thesis. We survey the literature to identify the different data management systems available as a part of the Hadoop software stack. We compare and contrast the features of different data management systems such as Hive, HBase and Pig. We explain the different features of Hive, the data warehouse system used in this thesis. We survey the literature to identify different benchmarking standards suitable to benchmark Hive. We choose the TPC-DS benchmark for our experiments since it has a data generator that is capable of generating large scale relational data and it defines OLAP-style queries (filtering and aggregation).

We equip four Dell Poweredge machines with additional storage space necessary for the Hadoop distributed file system. We select the Hadoop distribution provided by the Hortonworks Distributed Platform (HDP 1.3.2) for running our experiments since it supports the Ambari-based automated Hadoop installation procedure. We configure the hardware and software and networking the machines based on the instruction manual specified by Hortonworks [HINST] with the assistance of Dr. Paul Talaga. The procedure followed to set up the hardware necessary for this thesis has been explained in Chapter 3. We select Query 7 from the TPC-DS benchmark to run our experiments since it supported both filtering and aggregation operations. We describe an experimental procedure that involved generating data related to Query 7, loading the data into the tables in both Hive and MySQL, and running Query 7 on both Hive and MySQL. After collecting the results obtained from running this experiment on datasets of different sizes, we analyze the

results obtained and conclude that for all the datasets used in the experiments, Hive executes Query 7 faster than MySQL and has linear scalability. For all the datasets used in the experiment, Hive also loads the larger datasets significantly faster than MySQL, while it is marginally slower when loading the smaller datasets. In other words, we identify the different areas where Hive performs better than MySQL and also identify areas where the MySQL performed better than Hive. We also identify some of the potential pitfalls with the Hive system such as the Metastore being a single point of failure. In the next section, we discuss different ideas for improving Hive such as adding more variety of data types, optimizing the performance of a Hive query by including column statistics of reducing the number of MapReduce jobs. We also discuss ideas for overcoming potential problems such as the Hive metastore being a single point failure.

### 5.2 Future Work

The research work done in this thesis can be extended further by increasing the size and variety of data and by using different benchmarking standards that are better suited to big data than TPC-DS. Benchmarks such as BigBench [BBNP13], which at the time of writing this thesis, are under development, would be the ideal choice for analyzing the efficiency of Hive. Research can also be extended by increasing the number of nodes in the cluster.

Chen et al. [CYW13] denormalize the TPC-DS schema to transform its snowflake schema into a star schema. They then format the data in this star schema into a MOLAP cube format, rather than a ROLAP format, which is used for relational tables. They compare the performance of their cube implementation with a Hive implementation, by using Query 7 from TPC-DS and some additional queries. While the details about the configuration and experiments have not been specified, Chen et al. indicate a speed up of their approach when compared to Hive for 1GB (14x speed up), 10 GB (24x speed up), and 100 GB (19x speed up). Since the details about the configurations and experimental procedures have not been elaborated, it is difficult to draw

conclusions with any confidence. However, the results do indicate that future experimentation with their approach may have significant benefits.

We anticipate that better results can be obtained in the future if research is conducted in these directions:

**a. More variety in data types**

In Chapter 4, we discussed the differences in HiveQL and SQL while defining a table. In the later releases of Hive [HIVE13] (versions 0.12 and 0.13) which were released after the completion of the experimental work for this thesis, some of these issues have been addressed. More data types such as VARCHAR and DATE have been defined in versions 0.12 and 0.13.

**b. Optimizing performance**

The performance of Hive can be optimized in several ways. Some potential areas have been discussed below:

- Optimizing the queries by storing statistical information in the form of metadata in the Metastore at column, partition, and table level as suggested by Gruenheid et al. [GOM11]. Grunheid et al. [GOM11] propose that storing more statistical information at the columnar level in the Metastore would help in improving a query execution plan by being able to determine several factors such as the order of joins, the number of MapReduce jobs to be run, and the number of rows that appear in the result.

- Optimizing the queries in order to lower the number of MapReduce jobs being generated. This has to be done since a MapReduce job is a batch process and as a result involves latency. As a result, running more MapReduce jobs increases the latency on the system.

- Optimizing the lower levels, such as optimizing the work flow of the MapReduce jobs. Research can be conducted into identifying means of efficiently placing data in different

intermediate stages of a MapReduce process to effectively use the massively parallel model of MapReduce. Models like StarFish [HLLB11] define a self-tuning system which adapts to user needs and optimizes the performance of a workflow of MapReduce jobs and acts as an intermediate between the Hive interface and the MapReduce programming model are a step in the right direction.

**c. Overcoming single point of failure issues with Metastore**

The Metastore that is used to store the metadata related to Hive tables is a potential single point of failure. Different back up or check point based mechanisms to overcome this issue can be further investigated.

In summary, this thesis achieves its research objectives in identifying a parallel data management system that is suitable for performing OLAP-type analysis (aggregation and filtering) of large scale relational data and using a benchmark to compare the performance of the parallel and relational data management systems for the aforementioned OLAP-type query. However, one limitation with the current Hive version is that it supports only equi-joins [CWR12].

## References


**[CWR12]** Capriolo, Edward, Dean Wampler, and Jason Rutherglen. *Programming hive*. O'Reilly Media, Inc., 2012.

**[CYW13]** Chen, Dan, Xiaojun Ye, and Jianmin Wang. "A multidimensional data model for TPC-DS benchmarking." In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, p. 21. ACM, 2013.

**[SD08]** Dewitt and Stonebraker's "MapReduce: A major step backwards" by Craig Henderson Available at http://craig-henderson.blogspot.com/2009/11/dewitt-and-stonebrakers-mapreduce-major.html

**[AAMB]** Apache Ambari. Available at http://ambari.apache.org/

**[PIG14]** Apache Pig homepage. Available at http://pig.apache.org/

**[SQOOP]** Apache Sqoop Home page. Available at http://sqoop.apache.org/

**[BBNP13]** Baru, Chaitanya, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. "Setting the direction for big data benchmark standards." In *Selected Topics in Performance Evaluation and Benchmarking*, pp. 197-208. Springer Berlin Heidelberg, 2013.

**[DG08]** Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51, no. 1 (2008): 107-113.

**[HIVE13]** Different data types on Hive. Available at https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types

**[DSGEN]** DSGen v1.1.0, data generation tool for TPC-DS. Available at http://www.tpc.org/tpcds/

**[GTPCDS13]** Generating data using TPC-DS. Available at http://www.innovationbrigade.com/index.php?module=Content&type=user&func=display&tid=1&pid=3

**[GRHR13]** Ghazal, Ahmad, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. "BigBench: Towards an Industry Standard Benchmark for Big Data Analytics." (2013).

**[J08]** Greg Jorgensen, "Relational Database Experts Jump The MapReduce Shark". Available at http://typicalprogrammer.com/?p=16

**[GMIX]** GridMix program. Available in Hadoop source distribution: src/benchmarks/gridmix

**[GOM11]** Gruenheid, Anja, Edward Omiecinski, and Leo Mark. "Query optimization using column statistics in hive." In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pp. 97-105. ACM, 2011.

**[TSORT]** Hadoop TeraSort program. Available in Hadoop source distribution since 0.19 version: src/examples/org/apache/hadoop/examples/terasort

**[HLLB11]** Herodotou, Herodotos, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. "Starfish: A Self-tuning System for Big Data Analytics." In *CIDR*, vol. 11, pp. 261-272. 2011.

**[HSERDE]** Hive documentation on SerDe. Available at https://cwiki.apache.org/confluence/display/Hive/SerDe

**[HPB]** Hive Performance Benchmark. Available at https://issues.apache.org/jira/browse/hive-396

**[HHDP]** Hortonworks HDP 1.3.2 configuration. Available athttp://hortonworks.com/products/hdp/hdp-1-3/#overview

**[HINST]** Hortonworks HDP 1.3.2 installation manual. Available at

http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.2/bk_using_Ambari_book/content/ambari-chap1.html

**[HSTNGR13]** Hortonworks Stinger Initiative. Available at http://hortonworks.com/labs/stinger/

**[HHDX10]** Huang, Shengsheng, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis." In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41-51. IEEE, 2010.

**[J09]** Jacobs, Adam. "The pathologies of big data." *Communications of the ACM* 52, no. 8 (2009): 36-44.

**[NP06]** Nambiar, Raghunath Othayoth, and Meikel Poess. "The making of TPC-DS." In *Proceedings of the 32nd international conference on Very large data bases*, pp. 1049-1058. VLDB Endowment, 2006.

**[PZ10]** Pansare, Niketan, and Zhuhua Cai. "Using Hive to perform medium-scale data analysis." (2010).

**[PPRA09]** Pavlo, Andrew, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. "A comparison of approaches to large-scale data analysis." In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 165-178. ACM, 2009.

**[RTPCH]** Running the TPC-H benchmark on Hive. Available at https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf

**[HSORT]** Sort program. Available in Hadoop source distribution: src/examples/org/apache/hadoop/examples/sort

**[SMZH10]** Shi, Yingjie, Xiaofeng Meng, Jing Zhao, Xiangmei Hu, Bingbing Liu, and Haiping Wang. "Benchmarking cloud-based data management systems." In *Proceedings of the second international workshop on Cloud data management*, pp. 47-54. ACM, 2010.

**[TSJS09]** Thusoo, Ashish, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive: a warehousing

solution over a map-reduce framework." *Proceedings of the VLDB Endowment* 2, no. 2 (2009): 1626-1629.

**[TPCC]** TPC-C benchmarking standard. Available at http://www.tpc.org/tpcc/

**[TPCDS]** TPC-DS documentation. Available at http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf

**[TPCH]** TPC-H benchmarking standard. Available at http://www.tpc.org/tpch/

**[TNHDP]** Types of nodes on HDP. Available at http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.2/bk_getting-started-guide/content/ch_hdp1_getting_started_chp3.html

**[W12]** White, Tom. *Hadoop: the definitive guide*. O'Reilly, 2012.

## Appendix A: Instructions for Generating Data Using TPC-DS

1. Download the TPC-DS package from the TPC-DS website [DSGEN].
2. Unzip the package and under the folder *tools* you will find the word document - “HOW TO GUIDE” which contains the necessary instructions to compile the code and then use the commands to generate data.
3. Further instructions for different ways data can be generated is located in the word document – “QGEN” which contains the help manual for different commands and extensions provided by TPC-DS
4. **Generating Data:**
    a. The HOW TO GUIDE contains the commands that can be used for generating the data. For example,

    $ dbgen2 –scale 50 –dir 50gb –force

    **Note 1:** *-scale* extension is used to specify the scale factor. In this case, we are generating 50GB worth data.

    **Note 2:** *-dir* is used to mention the directory in which the data is to be stored. In this example, we have a directory named 15 GB where the data is being stored.

    **Note 3:** *-force* is used to ensure that old files in the *–dir* folder are overwritten. If the destination folder (*-dir*) is not empty, unless *-force* is used, it will cause an error.

    b. In case you have issues using the “dbgen2” command as shown in the HOW TO GUIDE, I would recommend using the following procedure[GTPCDS13]:

    After using the *make* command in Step 3 in the HOW TO GUIDE, if you face problems with the system being unable to recognize the dbgen2 command, you can use the following command to generate data:

$ ./dsdgen -scale 50 -dir 50gb -force

5. **Generating Queries:**

   a. The HOW TO GUIDE contains the commands to generate the SQL code from the templates present in the *query_templates* folder.

      $qgen2 –query10.tpl –directory query_templates –dialect oracle –scale 100

      Note 1: The .tpl file is the common template file which gets modified by the qgen command into a sql file.

      Note 2: The exact SQL syntax into which the template file (in this example, query10.tpl) is converted to is determined by the dialect. In this example, the oracle dialect is used, since it follows a similar syntax to MySQL.

      Note 3: The directory in which the template file is located is mentioned using –*directory* extension.

      Note 4: The –*scale* option is used to specify the scale of the data to which the query is being generated. In this example, the scale is 100 which means that 100GB of data has been generated.

      Note 5: The output of this command is stored in a file named query_0.sql in the same folder where this query is being run.

   b. In case you have issues with the qgen2 command shown above, copy the *query_templates* folder into the *tools* folder and then use the following command:

      $./dsqgen -input query_templates/templates.lst -directory query_templates -scale 100 -dialect oracle

Note 1: –*directory, -scale,* and –*dialect* function the same way as mentioned above. The –*input* option allows us to specify what templates we are attempting to convert into sql.

The *templates.lst* file contains the names of all the queries we wish to convert into sql, with one query name per line.

Note 2: While running the above command, one possible error that might occur is as shown below:

```
Substitution'_END' is used before being initialized

 query_templates/query7.tpl
```

The solution to this error is to add the line shown below at the beginning of the tpl file of the query.

```
define _END="";
```

## Appendix B: Additional Results

In this Appendix, we display some additional results of experimental procedure specified in Chapter 4.

1. **Dataset 4: Total data set size:** 20 GB

   **Total data set size for the query:** 7.5 GB

   Table 4.7.1 contains the sizes and the number of rows in each dataset. Table 4.7.2 displays the amount of time taken for defining the schema, loading the data and to execute the query in both MySQL and Hive.

**Table 4.7.1** *Total size and number of records in datasets used for the query-Dataset 4*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 7.6 MB | 28,000 |
| Promotion | 43 KB | 355 |
| store_sales | 7.5 GB | 57,598,932 |
| date_dim | 9.9 MB | 73,049 |

**Table 4.7.2** *Table definition, data loading and query execution times--Dataset 4*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.12s | 0.502s | 5.9s | 2.44s | 4m43.64s | 1m1.783s |
| Item | 0.11s | 0.195s | 0.26s | 1.251s | | |
| Promotion | 0.13s | 0.137s | <0.01s | 0.527s | | |
| store_sales | 0.10s | 0.171s | 2h17m13.3s | 3m45.436s | | |
| date_dim | 0.11s | 0.161s | 0.4s | 0.682s | | |

2. **Dataset 5: Total data set size:** 30 GB

   **Total data set size for the query:** 12 GB

   Table 4.8.1 contains the sizes and the number of rows in each dataset. Table 4.8.2 displays the amount of time taken for defining the schema, loading the data and to execute the query in both MySQL and Hive.

**Table 4.8.1** *Total size and number of records in datasets used for the query-Dataset 5*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 11 MB | 40,000 |

| | | |
|---|---|---|
| Promotion | 50 KB | 411 |
| store_sales | 12 GB | 86,406,277 |
| date_dim | 9.9 MB | 73,049 |

**Table 4.8.2** *Table definition, data loading and query execution times-Dataset 5*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.10s | 0.35s | 5.88s | 3.019s | 7m29.68s | 1m20.945s |
| Item | 0.10s | 0.137s | 0.36s | 0.775s | | |
| Promotion | 0.11s | 0.168s | 0.02s | 0.486s | | |
| store_sales | 0.10s | 0.157s | 3h50m45.85s | 6m34.599s | | |
| date_dim | 0.12s | 0.154s | 0.37s | 1.016s | | |

3. **Dataset 6: Total data set size: 40 GB**

   **Total data set size for the query: 16 GB**

   Table 4.9.1 contains the sizes and the number of rows in each dataset. Table 4.9.2 displays the amount of time taken for defining the schema, loading the data and to execute the query in both MySQL and Hive.

**Table 4.9.1** *Total size and number of records in datasets used for the query-Dataset 6*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 14 MB | 52,000 |
| Promotion | 57 KB | 466 |
| store_sales | 15 GB | 115,203,420 |
| date_dim | 9.9 MB | 73,049 |

**Table 4.9.2** *Table definition, data loading and query execution times-Dataset 6*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.11s | 0.331s | 5.92s | 2.658s | 10m9.68s | 1m43.585s |
| Item | 0.11s | 0.155s | 0.46s | 1.544s | | |
| Promotion | 0.10s | 0.135s | 0.01s | 0.488s | | |
| store_sales | 0.10s | 0.172s | 6h21m10.38s | 8m55.387s | | |
| date_dim | 0.11s | 0.174s | 0.47s | 0.798s | | |

4. **Dataset 7: Total data set size: 60 GB**

   **Total data set size for the query: 23 GB**

   Table 4.10.1 contains the sizes and the number of rows in each dataset. Table 4.10.2 displays the amount of time taken for defining the schema, loading the data and to execute the query in both MySQL and Hive.

**Table 4.10.1** *Total size and number of records in datasets used for the query-Dataset 7*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 20 MB | 74,000 |
| Promotion | 71 KB | 577 |
| store_sales | 23 GB | 172,800,711 |
| date_dim | 9.9 MB | 73,049 |

**Table 4.10.2** *Table definition, data loading and query execution times-Dataset 7*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | MySQL | Hive | MySQL | Hive | MySQL | Hive |
| customer_demographics | 0.11s | 0.33s | 5.83s | 2.817s | 5m16.34s | 2m23.65s |
| Item | 0.09s | 0.134s | 0.64s | 1.955s | | |
| Promotion | 0.10s | 0.174s | 0.02s | 0.55s | | |
| store_sales | 0.10s | 0.145s | 8h26m38.7s | 12m28.502s | | |
| date_dim | 0.11s | 0.161s | 0.47s | 0.765s | | |

5. **Dataset 8: Total data set size: 80 GB**

   **Total data set size for the query: 31 GB**

   Table 4.11.1 contains the sizes and the number of rows in each dataset. Table 4.11.2 displays the amount of time taken for defining the schema, loading the data and to execute the query in both MySQL and Hive.

**Table 4.11.1** *Total size and number of records in datasets used for the query-Dataset 8*

| Table | Size | Number of rows |
|---|---|---|
| customer_demographics | 77 MB | 1,920,800 |
| Item | 26 MB | 96,000 |
| Promotion | 84 KB | 688 |
| store_sales | 31 GB | 230,394,794 |
| date_dim | 9.9 MB | 73,049 |

**Table 4.11.2** *Table definition, data loading and query execution times-Dataset 8*

| Table | Table definition | | Data Load | | Query Execution | |
|---|---|---|---|---|---|---|
| | **MySQL** | **Hive** | **MySQL** | **Hive** | **MySQL** | **Hive** |
| customer_demographics | 0.16s | 0.301s | 5.87s | 2.63s | 10m36.68s | 3m10.484s |
| Item | 0.11s | 0.155s | 0.82s | 1.395s | | |
| Promotion | 0.11s | 0.154s | 0.02s | 0.536s | | |
| store_sales | 0.15s | 0.164s | 13h55m9.15s | 16m35.423s | | |
| date_dim | 0.14s | 0.154s | 0.38s | 0.7s | | |