Boise State University
ScholarWorks

Computer Science Graduate Projects and Theses | Department of Computer Science

8-1-2012

Hadoop and Hive as Scalable Alternatives to RDBMS: A Case Study

Marissa Rae Hollingsworth
Boise State University

HADOOP AND HIVE AS SCALABLE ALTERNATIVES TO RDBMS: A CASE STUDY

by

Marissa Rae Hollingsworth

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2012

© 2012
Marissa Rae Hollingsworth
ALL RIGHTS RESERVED

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the project submitted by

Marissa Rae Hollingsworth

Project Title: Hadoop and Hive as Scalable alternatives to RDBMS: A Case Study

Date of Final Oral Examination: 13 August 2012

The following individuals read and discussed the project submitted by student Marissa Rae Hollingsworth, and they evaluated their presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain	Chair, Supervisory Committee
Murali Medidi	Member, Supervisory Committee
Jyh-haw Yeh	Member, Supervisory Committee

The final reading approval of the project was granted by Amit Jain, Chair, Supervisory Committee. The project was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

For Nathan.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Amit Jain, for the dedication and encouragement he gave during both my undergraduate and graduate studies at Boise State University. I would also like to thank my committee members, Dr. Murali Medidi and Dr. Jyh-haw Yeh, Dave Arnett and Vince Martino for the guidance they provided. And finally, I would like to thank my family and friends for their encouragement, love and support; I could not have done this without them.

AUTOBIOGRAPHICAL SKETCH

Marissa Hollingsworth was born in Carlsbad, California and moved to Boise, Idaho in 2004. She graduated from Skyview High School in 2004 and attended Boise State University for her undergraduate career. She graduated from Boise State with a Bachelor of Science degree in Computer Science in 2009 and began her graduate studies at Boise State University immediately after, working as a graduate teaching assistant for the computer science department. In June of 2011, she began her career as a Software Design Engineer at HP Indigo in Boise, where she is currently employed.

ABSTRACT

While high-performance, cost-effective data management solutions, such as Hadoop, exist for Big Data analysis, small and medium businesses with moderate-sized data sets would also like to implement low budget data management systems that will perform well on existing data and scale as the amount of accumulated data increases. Parallel database management systems may provide a high-performance solution, but are expensive and complex to implement. The purpose of this project was to compare the scalability of open-source relational database management systems and distributed data management systems for small and medium data sets. To make this comparison, a business intelligence case study was investigated using three data management solutions: MySQL, Hadoop MapReduce, and Hive. This experiment involved a payment history analysis which considers customer, account, and transaction data for predictive analytics. Experiments were executed on data sets ranging from 200MB to 10GB. The results show that the single server MySQL solution performs best for trial sizes ranging from 200MB to 1GB, but does not scale well beyond that. MapReduce outperforms MySQL on data sets larger than 1GB and Hive outperforms MySQL on sets larger than 2GB. This demonstrates MapReduce and Hive as viable techniques for small and medium businesses who want to implement scalable data management techniques.

TABLE OF CONTENTS

ABSTRACT vii

LIST OF TABLES xi

LIST OF FIGURES xii

1 Introduction 1

1.1 Overview 1

1.2 Structure 3

2 Background 5

2.1 Business Intelligence 5

2.2 Relational Database Management Systems 7

2.2.1 MySQL 7

2.3 Big Data Analytics 8

2.3.1 Hadoop Distributed Filesystem 9

2.3.2 MapReduce 9

2.3.3 Hive 10

3 Payment History Analysis: A Case Study and Problem Statement 12

3.1 The Case Study 13

3.1.1 Data Storage and Relationship Model 13

3.1.2 Data Access Model 15

3.1.3 Account Data Generation 16
3.2 Problem Statement 17

4 Design and Implementation 20
4.1 Test Data Generation Phase 20
4.1.1 HistogramGen: Statistical Analysis of Sample Data 21
4.1.2 AccountGen: Test Data Generation 25
4.2 MySQL Solution 26
4.3 Hadoop MapReduce Solution 27
4.3.1 Stage 1: Customer Join Account 27
4.3.2 Stage 2: Strategy History Join Transaction 29
4.3.3 Stage 3: Combine Results 33
4.4 Hive Solution 34

5 Experimental Results and Analysis: MySQL, MapReduce, and Hive Performance Comparison 35
5.1 Setup 35
5.2 Execution 37
5.3 Results 38

6 Conclusion 42
6.1 Summary 42
6.2 Results and Implications 43
6.3 Future Direction 44

REFERENCES 45

A MySQL Payment History Database Schema 47

B Cluster Configuration 48

C Hive Payment History Database Schema 50

LIST OF TABLES

3.1 Customer Tuple 14

3.2 Account Tuple 14

3.3 Transaction Tuple 14

3.4 StrategyHistory Tuple 15

3.5 Expected Result Tuple - Attribute descriptions 19

5.1 Approximate trial size of data set for number of Account records 38

5.2 Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running on master node plus four data nodes 39

5.3 Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running in pseudo-distributed mode 40

LIST OF FIGURES

3.1 Account history data entity-relationship model 18

4.1 `HistogramGen` program flow diagram 21

4.2 Stage 1: Customer join Account program flow diagram 28

4.3 Stage 2: Stragety History join Transaction program flow diagram 30

4.4 Stage 3: Combine results program flow diagram 33

5.1 Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running on master node plus four data nodes 39

5.2 Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running on master node in pseudo-distributed mode 40

5.3 Scalability of MapReduce run-time (seconds) performance for a varied number of map tasks per datanode 41

B.1 Boise State University, Department of Computer Science, Onyx Cluster Lab 48

Chapter 1

INTRODUCTION

1.1 Overview

The persistent evolution of digital technology is unlocking new forms of Business Intelligence (BI). The acquisition, storage, access, and analysis of digital information provides knowledge and strategic insights for enterprise and research organizations. Each time we visit a website or make a transaction, the owner has the ability to log everything we do; every link we click, each form we submit, what time we log in and out, any errors we encounter, and just about anything else we can imagine. While this information may seem useless and doesn't appear to affect the user much, the site owner now has valuable information that can give him or her the competitive advantage. The collected data can be stored in log files and analyzed for BI that can be used to develop and incorporate new business strategies and direct planning for the future.

In terms of BI data management, collection is generally the easy part– it is the storage and access aspects (requisites for later BI analytics) that impose major challenges for *small and medium businesses* (SMB)– especially since data can accumulate to gigabytes and even terabytes in a relatively short period of time. Thus, data management becomes a greater obstacle as more and more data needs to be collected and stored. Traditionally, businesses have been able to store their data in local data centers comprised of a few machines. However, as data outgrows the capacity of local data centers, they must modify their storage

methods. Even if the company expands the data to multiple machines, they must develop ways of accessing the data. In BI, more data reveals new doors and opportunities for the organization in terms of derived knowledge and achieving a competitive advantage, but the information is effectively useless if they can't consider all of it simultaneously in their strategic analysis.

Providing a multi-layer model of staging, integration, and access to historical data archives is fundamental to BI for any modern organization. Staging is used to store raw data for use by developers, integration is used to incorporate data into the storage system while maintaining a level of abstraction from users, and the access layer is for getting data out for the end users. Due to the sensitive and irreplaceable nature of customer data, enterprise storage systems must provide data replication, backup and recovery, data integrity, and concurrency control. Common storage systems include data transformation algorithms to transform data to the correct format. The system interacts with the operating system to send the data from the user to the storage system. There are several database management systems which conform to these requirements, but the most popular is the *relational database management system* (RDBMS).

As the size of data sets increase, it is possible to run out of space on a single system. When this happens, the data may be moved to a parallel RDBMS or distributed among several networked systems and managed by a *distributed data management system* (DDMS). Parallel RDBMSs have been commercially available for several decades and offer high performance and high availability for data management. However, the up-front costs and the cost to scale up for larger storage capacities can be expensive. On the other hand, freely available DDMSs offer high performance and high availability, but at much lower up-front and scaling costs.

In this research, we investigate the performance of *open-source software* (OSS) so-

lutions on mid-sized data sets for SMBs using a realistic case study. We conduct this experiment between a RDBMS, namely MySQL, and a DDMS, namely Hadoop, which is a resource-effective solution for SMBs. We investigate MapReduce and Hive as two techniques for analyzing the data stored in Hadoop.

1.2 Structure

In Chapter 2 we provide the background for concepts pertaining to BI storage, access, and analytical systems. Here, we outline some challenges faced by SMBs interested in BI analytics and discuss RDBMS and alternative DDMS solutions. We summarize the notion of Big Data, a likely future obstacle for the SMB, and provide relevant information regarding the Hadoop solutions considered in this research: MapReduce and Hive.

In Chapter 3 we introduce the payment history analysis case study for a specific SMB. This particular SMB designs software tools to analyze current and historical payment habits on customer data and attempts to forecast trends. We discuss the database management challenges faced by the SMB as their customer base expands and define their specific case study. Here, we explain their data storage and access models, along with the requisite test data set generation utilities required to conduct the analysis. In doing so, we formally present the problem statement and list the inquiries we wish to address in this research project.

In Chapter 4 we define the design and implementation for the RDBMS and DDMS solutions. First, we discuss the detailed process of generating the test data set and provide a summarized description of the key algorithms and data structures employed by these utilities: the `HistogramGen` and `AccountGen` MapReduce programs. Second, we discuss the central components of the MySQL, Hadoop MapReduce, and Hadoop Hive solution

implementation, where we explain the process by which they are used to instrument the payment history analysis case study (as presented in Chapter 3).

In Chapter 5 we detail the RDBMS and DDMS performance comparison experiment (for the solutions from Chapter 4). Here, we discuss the efficiency and scalability for these implementations, along with the resulting implications and analysis for this SMB's data management case study (as expressed in Chapter 3). First, we discuss the software and hardware environment in which our comparative benchmark analysis is conducted. Second, we provide the procedure used to guide and execute the experiment. Finally, we present the performance results for the various experimental trial runs.

In Chapter 6 we conclude our report and suggest some future directions.

Chapter 2

BACKGROUND

2.1 Business Intelligence

BI aims to support decision-making for an organization. As digital information becomes more prevalent, service providers are seeking guidance as to how they can leverage the data they collect to improve their knowledge base for incorporating new strategies into their business. BI technologies provide historical, current and predictive views of business operations; BI software is designed to perform a wide variety of such functions, ranging from online analytical processing, to business performance management, to data, process, and text mining.

Until recently, the focus for data analysis had been on large enterprises with vast amounts of data. However, a major problem faced by SMBs in today's economy is their lack of management expertise and financial reporting that investors need to make decisions about whether or not to invest [11]. More than one-quarter of SMBs indicated that "getting better insights from the data" they have is a top technology challenge [1]. Overcoming this mode of challenges places crucial importance on the ability to implement a scalable BI solution based on the available data and thereby provide a means to acquire valuable insights into business trends.

In general, a BI system comprises three essential and distinct components, namely *data sources*, *data warehousing*, and *analytics*. Data may be sourced from a diverse set of

entities, which may be external (i.e. industry reports, media data, business partners, etc.) and/or internal (i.e. account transactions, employee reports, client reports, etc.). Data sets used for BI is typically defined as Big Data and may be unstructured and complex. Data warehousing is essential to an effective BI system and should be disjoint from the operational data storage required for day-to-day business operations. In this context, a data warehouse contains a historical data store designed to manage data accumulated over time and an analytical data store designed to manage and make available a subset of the historical store for predictive analysis. Finally, the analytics component includes the software tools which access the analytical data store and instrument the prediction procedure(s).

In the initial BI development stages it is typical for businesses to use a RDBMS for *both* operational and data warehousing. When the data sets are small, this strategy is beneficial because the approach is relatively simple, efficient, and cost-effective. However, as time progresses and accumulated data increases, this "one size fits all" approach does not work well with BI analysis, largely due to the underlying architecture of the database [9, 14]. "To achieve acceptable performance for highly order-dependent queries on truly large data, one must be willing to consider abandoning the purely relational database model for one that recognizes the concept of inherent ordering of data down to the implementation level" [9]. Distributed solutions designed to manage big data, such as the Apache Hadoop project, are adopted to meet these new requirements. The following sections provide background information on the RDBMS and big data analysis techniques used in our research– MySQL and Apache Hadoop, respectively.

2.2 Relational Database Management Systems

The relational database is the most popular database model used for the storage and access of operational data that is optimized for real-time queries on relatively small data sets. Data is organized as a set of formally-described tables whose fields are represented as columns and records are represented as rows in the table [3]. A RDBMS is the software which controls the storage, retrieval, deletion, security, and integrity of data within a relational database [2]. Data can be accessed and reassembled in several ways for both interactive queries and gathering data for reports via structured query language (SQL) operations based on relational algebra.

2.2.1 MySQL

MySQL Community Edition is the free version of "the world's most popular" open-source RDBMS implementations that is supported by a huge and active community of open source developers. MySQL supports many standard DBMS features including replication, partitioning, stored procedures, views, MySQL Connectors for building applications in multiple languages, and the MySQL Workbench for visual modeling, SQL development and administration [10]. Many organizations such as Facebook, Google, Adobe, and Zappos use MySQL to power high-volume web sites, business-critical systems and packaged software.

A typical MySQL deployment includes a server installed on a single, high-end server which accepts local and remote client queries. Since the database is limited to the hard drives of the server, when the amount of data exceeds the limited storage capacity, this model will fail and a DBMS with an underlying distributed storage system will need to be employed.

2.3 Big Data Analytics

While the RDBMS model is well suited and optimized for real-time queries on relatively small data sets, it was *not* designed for Big Data analysis, largely due to the limited storage capacities and the underlying write-optimized "row-store" architecture [14]. While write-optimization allows for efficient data import and updates, the design limits the achievable performance of historical data analysis that requires optimized read access for large amounts of data. Another drawback of the RDBMS approach stems from the lack of scalability as the number of stored records expands. To overcome this obstacle, we can move data to a parallel DBMS system.

Parallel DBMSs share the same capabilities as traditional RDMBSs, but run on a cluster of commodity systems where the distribution of data is transparent to the end user [12]. Parallel RDBMSs have been commercially available for several decades and offer high performance and high availability, but are much more expensive than single-node RDBMSs because there are no freely available implementations and they have much higher up-front costs in terms of hardware, installation, and configuration [13]. In contrast, Hadoop can can be deployed on a cluster of low-end systems and provides a cost-effective, "out-of-the-box" solution for Big Data analysis. While some parallel DBMSs may have relative performance advantages over open-source systems, such as Hadoop, the set-up cost and cost to scale may deter SMBs from using them. Furthermore, Hadoop is better suited for BI analysis because it allows for the storage and analysis of unstructured data, while parallel DBMSs force the user to define a database schema for structured data [12].

2.3.1 Hadoop Distributed Filesystem

Hadoop is an open-source Java implementation of the MapReduce framework developed by Google. The Apache Software Foundation and Yahoo! released the first version in 2004 and continue to extend the framework with new sub-projects. Hadoop provides several open-source projects for reliable, scalable, and distributed computing [5]. Our project will use the Hadoop Distributed Filesystem (HDFS) [6], Hadoop MapReduce [7], and Hive [8].

The Hadoop Distributed Filesystem (HDFS) is a scalable distributed filesystem that provides high-throughput access to application data [6]. HDFS is written in the Java programming language. A HDFS cluster operates in a master-slave pattern, consisting of a master *namenode* and any number of slave *datanodes*. The namenode is responsible for managing the filesystem tree, the metadata for all the files and directories stored in the tree, and the locations of all blocks stored on the datanodes. Datanodes are responsible for storing and retrieving blocks when the namenode or clients request them.

2.3.2 MapReduce

MapReduce is a programming model on top of HDFS for processing and generating large data sets which was developed as an abstraction of the *map* and *reduce* primitives present in many functional languages [4, 7]. The abstraction of parallelization, fault tolerance, data distribution and load balancing allows users to parallelize large computations easily. The map and reduce model works well for Big Data analysis because it is inherently parallel and can easily handle data sets spanning across multiple machines.

Each MapReduce program runs in two main phases: the map phase followed by the reduce phase. The programmer simply defines the functions for each phase and Hadoop handles the data aggregation, sorting, and message passing between nodes.There can be

multiple map and reduce phases in a single data analysis program with possible dependencies between them.

Map Phase. The input to the map phase is the raw data. A map function should prepare the data for input to the reducer by mapping the key to the the value for each "line" of input. The key-value pairs output by the map function are sorted and grouped by key before being sent to the reduce phase.

Reduce Phase. The input to the reduce phase is the output from the map phase, where the value is an iterable list of the values with matching keys. The reduce function should iterate through the list and perform some operation on the data before outputting the final result.

2.3.3 Hive

While the MapReduce framework provides scalability and low-level flexibility to run complex jobs on large data sets, it may take several hours or even days to implement a single MapReduce job [16]. Recognizing this, Facebook developed Hive based on familiar concepts of tables, columns and partitions, providing a high-level query tool for accessing data from their existing Hadoop warehouses [16]. The result is a data warehouse layer built on top of Hadoop that allows for querying and managing structured data using a familiar SQL-like query language, HiveQL, and optional custom MapReduce scripts that may be plugged into queries [8, 15]. Hive converts HiveQL transformations to a series of MapReduce jobs and HDFS operations and applies several optimizations during the compilation process.

The Hive data model is organized into *tables*, *partitions* and *buckets*. The *tables* are similar to RDBMS tables and each corresponds to an HDFS directory. Each table can be divided into *partition*s that correspond to sub-directories within an HDFS table directory

and each partition can be further divided into *buckets* which are stored as files within the HDFS directories [15].

It is important to note that Hive was designed for scalability, extensibility, and batch job handling, *not* for low latency performance or real-time queries. Hive query response times for even the smallest jobs can be of the order of several minutes and for larger jobs, may be on the order of several hours [8].

Chapter 3

PAYMENT HISTORY ANALYSIS: A CASE STUDY AND PROBLEM STATEMENT

The payment history analysis case study has been provided by a local software company who wishes to remain anonymous. From this point on, we will refer to this company as "CompanyX". CompanyX provides software tools for customer information management and record analysis for BI. CompanyX evaluates current and historical payment habits for each customer account and attempts to predict future payment patterns based on past trends. Successful prediction of whether a payment will be on-time, past-due, or delinquent may reduce administrative costs accrued by outsourcing delinquent accounts for payment collection. For example, if a customer has always made late payments in the past, they can expect that the customer will continue to make payments, even if they are late, and thus avoid sending them to a collection agency.

As they expand their services, CompanyX expects the number of customers to increase from several hundred to several *thousand*. This is a major cause for concern because the amount and complexity of collected data will increase as well, forcing the storage and access of data from their current hardware system to a more sophisticated system. Adopting a new hardware system while maintaining their existing service can be expensive, time-consuming, and may compromise data integrity and service availability as data is transferred from the old system to the new. Because of the risks involved, the company

would like to compare the viability of other systems before converting their entire business architecture. This project will address the two major concerns that arise as a consequence of the increasing data volume: *data storage model* and the *scalability of data analysis software*. Due to privacy rights, any records collected by the company cannot be used, so sample data sets must be generated as prerequisite for testing new system implementations.

Our research assesses potential open source data warehousing models for CompanyX.

3.1 The Case Study

3.1.1 Data Storage and Relationship Model

The data must be staged for data warehousing. During this process, the data is pruned and normalized to fit a chosen data storage model. In this case, CompanyX has selected an RDBMS for data management (deployed on a local, in-house server), which is used for both operational and warehousing purposes. For this, the data is normalized to include a(n):

1. `Customer` tuple for each customer (Table 3.1),
2. `Account` tuple for each customer account (Table 3.2),
3. `Transaction` tuple for each account transaction (Table 3.3), and
4. `StrategyHistory` tuple for each account strategy (Table 3.4).

Note that the referenced tables use a solid underline to identify the primary key and a dashed underline to identify a foreign key. The relationship model between the tables is described below and shown in Figure 3.1.

Customer: The unique *primary key* for the customer entity-type is the `CustomerNumber` attribute. Each customer entity stores the customer's social security number (`ssn`) and the first three digits of the customer's zip code (`zipcode3`).

Table 3.1: Customer Tuple

CustomerNumber	FirstName	LastName	Ssn	ZipCode3

Account: The unique *primary key* for the account entity-type is the `AccountNumber` attribute. Each account entity includes the date the account was opened (`opendate`) and is *is-owned-by* a unique customer, which is referenced by a foreign key (`CustomerNumber`).

Table 3.2: Account Tuple

AccountNumber	OpenDate	CustomerNumber

Transaction: The transaction entity is a weak type which depends on the account entity. Each transaction entity *is-owned-by* a unique account which is referenced by the `AccountNumber` key and includes the type of transaction (`TransactionType` $\in$ {`charge`, `adjustment`, `payment`}), the date of the transaction (`TransactionDate`), and the amount of the transaction (`TransactionAmount`). The unique identifier is a *composite key* composed of the `AccountNumber` reference, `TransactionType`, `TransactionDate`, and `TransactionAmount`.

Table 3.3: Transaction Tuple

AccountNumber	TransactionType	TransactionDate	TransactionAmount

StrategyHistory: The strategy history entity-type also depends on the account entity. Each strategy history entity *is-owned-by* a unique account which is referenced by the `AccountNumber` key and includes the name of the strategy (`StrategyName` $\in$ {`Good Standing`, `Bad Debt`}) and the start date of the strategy (`StrategyStartDate`). The unique identifier is a *composite key* composed of the `AccountNumber` reference, `StrategyName`, and `StrategyStartDate`.

Table 3.4: StrategyHistory Tuple

AccountNumber	StrategyName	StrategyStartDate

3.1.2 Data Access Model

In order for CompanyX to use the data as input for their payment prediction algorithms, they must extract and prepare *meaningful* data from the database. Data retrieval is achieved through join, aggregate, and other relevant SQL queries to the database. The resulting output of these queries must supply the input parameters for the prediction algorithms:

- Aggregate charge, adjustment, and payment sums of historical transaction data.
- Aggregate payment sums of historical transaction data for all previous accounts owned by the same customer.

For a complete summary of CompanyX's expected output see Table 3.5.

The first step of the retrieval process is to gather the transaction history data for each account, per strategy period. Recall that the relationships between account and transaction and customer and account are of *one-to-many*. The total charges and adjustments do not depend on the strategy period, so these fields only require data stored in the `Transaction` table. However, because the account payments are subdivided into thirty, sixty and ninety day good standing and bad debt strategy periods, both the `StrategyHistory` and `Transaction` tables are required to select transactions per period. The process can be broken into these simplified steps:

1. Join the `Account` and `Customer` tables to associate customer data with each account.
2. Join the `Account` and `Transaction` tables to collect aggregate sums of `charge` and `adjustment` transactions, per account.
3. Join the `Account`, `StrategyHistory` and `Transaction` tables to collect aggregate sums of `payment` transactions by strategy history type.

At this point, the attributes stored in the result entity for an account include the customer data, total charges and adjustments, and payments per strategy period. The next step will aggregate the transactions for all previous accounts with the same `ssn`. The result depends on the `OpenDate` of each account with the same `CustomerNumber` to determine if it is a previous account. If the account was opened before the current account, then the result will depend on the `Transaction` and `StrategyHistory` tables. The process can be broken into these simplified steps:

1. Determine the `ssn` of each account by joining `Account` and `Customer`.
2. Left outer join `Account` table with itself on `ssn` and select accounts where `OpenDate` of the joined account is before the `OpenDate` of the left outer account.
3. Join the result of the previous step with `StrategyHistory` and `Transaction` tables to collect aggregate sums of previous account transactions by transaction type, strategy history type, and account.

3.1.3 Account Data Generation

To respect customer privacy, potential test data related to real-world customer accounts and transaction histories have been withheld. In order to provide us with a starting point, CompanyX used their expert knowledge of the data trends to compile a small sample of transactions, strategy histories, and customer data for 200 accounts. However, several thousand accounts are needed for an accurate comparison between the performance metrics of potential solutions and the current implementation. Therefore, we must generate our own test data. To maintain quality as the quantity increases, the attributes of the generated data sets should adhere to the same probability distributions as the sample data. More detailed information about these attributes are outlined in Chapter 4.

3.2 Problem Statement

CompanyX is faced with challenges pertaining to Big Data:

1. There is too much data to store on single machine.
2. The database access times increase drastically as the amount and complexity of data accumulates over time.

For this, we wish to know if we can use distributed model, such as Hadoop, without significant loss of performance on existing *small* data sets, in order to prepare for the future *big* data sets. In doing so, perhaps it is possible to achieve a performance gain on current and future data sets. So we know Hadoop performs well for Big Data analysis, but how will it perform in this particular case?

Thus, the goals of the research project for CompanyX are to:

- Design and implement a scalable BI solution for extracting patterns in customer history data using existing open-source projects.
- Generate large sample data sets (hundreds of gigabytes) using Hadoop to compare the scalability of the MapReduce solution to the existing MySQL solution.
- Implement and compare a Hadoop MapReduce solution.
- Implement and compare a Hadoop Hive solution.

Therefore, we will address the following questions:

1. At what scale of data (number of accounts or customers) does each solution outperform the others?
2. Will the existing data schema work with each solution?
3. How much cost and effort will it take to deploy each solution?

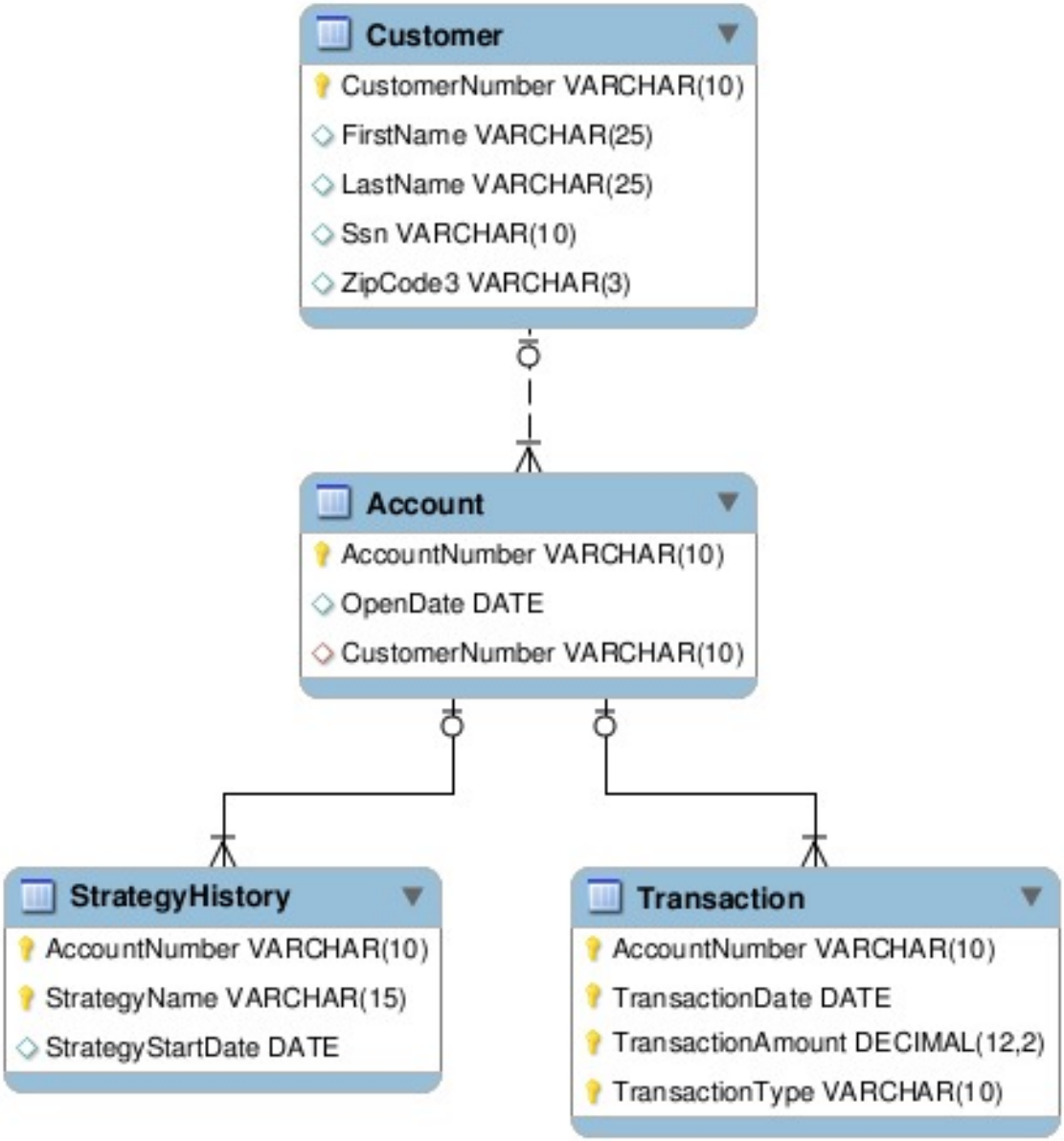

Figure 3.1: Account history data entity-relationship model

Table 3.5: Expected Result Tuple - Attribute descriptions

AccountNumber	unique Account identifier
OpenDate	date the Account was opened by Customer
ZipCode3	first three digits of Customer's zip code
TotalCharges	sum of all Charge transactions on Account
TotalAdjustments	sum of all Adjustment transactions on Account
AdjustedTotalCharges	sum of TotalCharges and TotalAdjustments
TotalGoodStandingPayments30Day	negative of sum of all payment transactions on account between 1 and 30 days after Good Standing strategy start date but before Bad Debt strategy start date
TotalGoodStandingPayments60Day	same as above, but between 1 and 60 days
TotalGoodStandingPayments90Day	same as above, but between 1 and 90 days
BadDebtTransferBalance	sum of all transactions on account up through "Bad Debt" strategy start date
TotalBadDebtPayments30Day	negative sum of all payment transactions on account between 1 and 30 days after Bad Debt strategy start date
TotalBadDebtPayments60Day	same as above, but between 1 and 60 days
TotalBadDebtPayments90Day	same as above, but between 1 and 90 days
PreviousAccountCount	number of accounts with open date prior to this account's open date which have a customer with the same SSN
PreviousAccountGoodStandingCharges	sum of all charge transactions occurring prior to this account's open date on accounts which have a customer with the same SSN (not same account) which occurred while other accounts were at or after Good Standing strategy start date but before other accounts were at Bad Debt strategy start date
PreviousAccountGoodStandingAdjustments	sum of all adjustments that occurred during Good Standing strategy for past accounts
PreviousAccountGoodStandingPayments	sum of all Payments that occurred during Good Standing strategy for past accounts
PreviousAccountBadDebtPayments	sum of all Payments that occurred during Bad Debt strategy for past accounts

Chapter 4

DESIGN AND IMPLEMENTATION

The developmental process for the design and implementation of our solutions to the payment history analysis case study is split into two distinct phases: *test data generation* and *solution implementation*. First, we generate our test data using Hadoop MapReduce, which is requisite to benchmarking each solution we implement. Second, we implement the RDBMS solution as a control in MySQL, followed by two DDMS solutions: Hadoop MapReduce and Hive.

4.1 Test Data Generation Phase

Before we can evaluate the scalability of each implementation, we need to generate large data sets based on the sample data provided by CompanyX. For each solution, the performance depends highly on the complexity of the relationships between customer, account, and transaction data. Therefore, it is crucial to maintain the probability distribution relationships of the sample data so we may achieve accurate and comparable benchmark results. For this, we introduce a package of test data generation utilities.

First, the `HistogramGen` MapReduce program identifies the probability distribution relationships for a subject sample data set. Second, the `HistogramGen` output is supplied as input to the `AccountGen` utility, which generates the test data sets.

4.1.1 HistogramGen: Statistical Analysis of Sample Data

The `HistogramGen` utility determines the probability distribution of the sample data set and generates a histogram report. Since performance depends on relationship complexity between account and customer data, we evaluate averages based on the number of:

1. accounts per customer,
2. customers per SSN,
3. charge, adjustment, and payment transactions per account, and
4. charge, adjustment, and payment transactions per strategy history.

Figure 4.1 provides an overview the `HistogramGen` program flow described below.

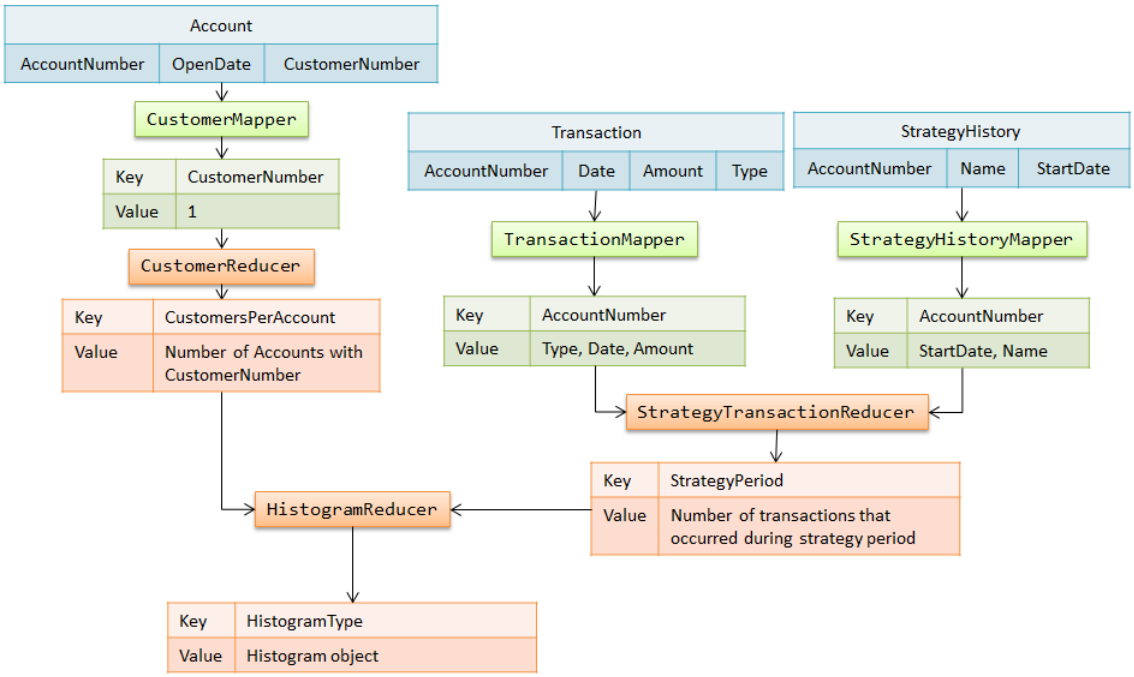

Figure 4.1: `HistogramGen` program flow diagram

The number of accounts per customer is determined by counting the number of `Account` entries with the same `CustomerNumber` attribute. This sum is found with the simple map

and reduce methods shown below. The CustomerMapper takes each Account entry as input and maps the CustomerNumber for the account to a value of 1.

```
public void map(Object offset, Text input,
     OutputCollector<Text, IntWritable> output, Reporter reporter)
     throws IOException
{
    String[] parts = (input.toString()).split(",");
    String customerNumber = parts[COL_CUSTOMER_NUM].trim();
    output.collect(new Text(customerNumber), ONE);
}
```

The CustomerReducer receives the mapped output for each CustomerNumber key from the CustomerMapper and sums the value of each to get the total number of accounts for the CustomerNumber key. The resulting output is the total number of accounts for the customer mapped to the histogram type.

```
public void reduce(Text customerNumber, Iterator<IntWritable> values,
      OutputCollector<IntWritable, IntWritable> output,
      Reporter reporter) throws IOException
{
    int count = 0;
    while (values.hasNext())
    {
        count += values.next().get();
    }
    /* output the type of histogram and the count */
    output.collect(histogramType, new IntWritable(count));
}
```

Furthermore, since performance depends on relationship complexity between the transaction data, the number of queries also depends on the strategy history and type of transaction for each strategy. Therefore, we evaluate the number of good standing and bad debt charges, adjustments, and payments per account. For each account, the number of

transactions per strategy history are determined by counting the number of `Transaction` entries with the same `AccountNumber` attribute and the number of `Transactions` that occur during each `StrategyHistory` period. These sums are found with the simple map and reduce methods shown below.

The `TransactionMapper` maps the `Transaction` to the owning `AccountNumber`.

```
public void map(Object offset, Text input,
     OutputCollector<Text, ObjectWritable> output, Reporter reporter)
     throws IOException
{
    String[] parts = input.toString().split(",");
    String account = parts[COL_ACCOUNT_ID].trim();
    String date = parts[COL_DATE].trim();
    String amount = parts[COL_AMOUNT].trim();
    String type = parts[COL_TYPE].trim();

    /* Output transactions by accountNumber */
    Transaction transaction = new Transaction(date, type, amount);
    output.collect(new Text(account), new ObjectWritable(transaction));
}
```

The `StrategyMapper` maps the `StrategyHistory` to the owning `AccountNumber`.

```
public void map(Object offset, Text input,
    OutputCollector<Text, ObjectWritable> output, Reporter reporter)
    throws IOException
{
    String[] parts = (input.toString()).split(",");
    String account = parts[COL_ACCOUNT_ID].trim();
    String name = parts[COL_NAME].trim();
    String startDate = parts[COL_DATE].trim();

    /* Output strategy history by accountNumber */
    StrategyHistory strategy = new StrategyHistory(startDate, name);
    output.collect(new Text(account), new ObjectWritable(strategy));
}
```

The `StrategyTransactionReducer` accepts key/value pairs from the `TransactionMapper` and `StrategyHistoryMapper`, iterates through the list of events for the key account, and sums the values for each account. The output is the number of each transaction type per `AccountNumber`.

```
public void reduce(Text key, Iterator<ObjectWritable> values,
        OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
        throws IOException
{
    /* Keep list of payments */
    List<EventWritable> events = new ArrayList<EventWritable>();

    /* Keep track of strategy dates */
    Calendar goodStrategy = new GregorianCalendar();
    Calendar badDebt = new GregorianCalendar();

    while (values.hasNext())
    {
        EventWritable next = (EventWritable)
        values.next().get();
        String type = next.getType().toString();
        events.add(next);
        /* Set good standing dates */
        if (type.equals(StrategyHistory.GOOD_STANDING))
        {
          //create event for 30, 60, 90 day after strategy starty date
        }
        /* Set bad debt dates */
        else if (type.equals(StrategyHistory.BAD_DEBT))
        {
          //create event for 30, 60, 90 day after strategy starty date
        }
    }

    /* Sort by date and find sums */
    Collections.sort(events);
    for(EventWritable event : events)
    {
        // count transactions per event strategy
    }
```

```
    // output one for each event strategy
    for(int i = 0; i < strategy_counts.length; i++)
    {
        output.collect(new IntWritable(i),newIntWritable(strategy_counts[i]));
    }
 }
```

Finally, the `HistogramReducer` consolidates the resulting output from the `CustomerReducer` and `StrategyTransactionReducer` into the final histogram.

```
public void reduce(IntWritable key, Iterator<IntWritable> values,
        OutputCollector<Text, Histogram> output, Reporter reporter)
        throws IOException
{
    /* What type of histogram is this? */
    HistogramType type = HistogramType.getType(key.get());

    /* add all the values to a temp list */
    ArrayList<Integer> temp = new ArrayList<Integer>();
    while (values.hasNext())
    {
        temp.add(values.next().get());
    }

    /* sort list and create a new Histogram from data */
    Collections.sort(temp);
    Histogram histogram = new Histogram(type, temp.toArray(new Integer[0]));
    output.collect(new Text(type.toString()), histogram);
}
```

4.1.2 AccountGen: Test Data Generation

The `AccountGen` program uses the `HistogramGen` probability distribution results to generate the test data sets with the `Customer`, `Account`, `Transaction`, and `StrategyHistory` entries.

To decrease time requirements for generating large data sets, the work load is distributed among several mappers. For this, we define our own `InputSplit` and `RecordReader` to send an account number range to each mapper. Then for each account, the mapper uses the histogram statistics to determine:

1. which `CustomerNumber` to assign, and
2. how many of each transaction type to generate (with the account number as the primary key).

The `MultipleOutputFormat` provided in the MapReduce API is used to output each entry (customer, account, transaction, strategy history) to a separate, comma-separated file. The attributes remain the same as the original entries.

4.2 MySQL Solution

Since CompanyX already has a RDBMS implementation of the payment history analysis case study in place, we were able to leverage the existing SQL script used to pull expected results for the desired output file from tables in our MySQL solution. We developed the MySQL solution in the following steps:

1. Define schema for payment history analysis database.
2. Implement MySQL script to load account, customer, transaction and strategy history data into database tables.
3. Update existing SQL script to execute in MySQL.

We were able to derive the table definitions, data types, and constraints for the database schema from the provided background data and SQL query (see Appendix A for complete schema). Since we will be generating new data sets for each benchmark execution, we need to be able to dynamically load data into the MySQL database. Thus, we implemented

a script to generate the MySQL load script based on a given input directory. The final load script is a series of statements of the following form– `'load data local infile <filename> into table <table name>'`. Finally, upon executing the SQL script in MySQL, we quickly realized that the existing SQL script used to pull expected results was not fully compatible with MySQL. So, the final step was to transform the incompatible queries to the MySQL standard and ensure that the result table matched the expected output table.

4.3 Hadoop MapReduce Solution

The MapReduce solution requires three map and reduce phases that are detailed in the following sections.

4.3.1 Stage 1: Customer Join Account

The first stage combines the Customer and Account tables on the CustomerNumber using a `CustomerMapper`, `AccountMapper` and `CustomerJoinAccountReducer` (see Figure 4.2).

Customer Mapper

- The input is the `Customer` data (formatted as a .csv file).
- The output key is `CustomerNumber` and the sort order for the key is `0`.
- The output value is `Ssn` and `ZipCode3` of the customer.

```
public void map(Object offset, Text input,
    OutputCollector<TextPair, Text> output, Reporter reporter)
    throws IOException {
```

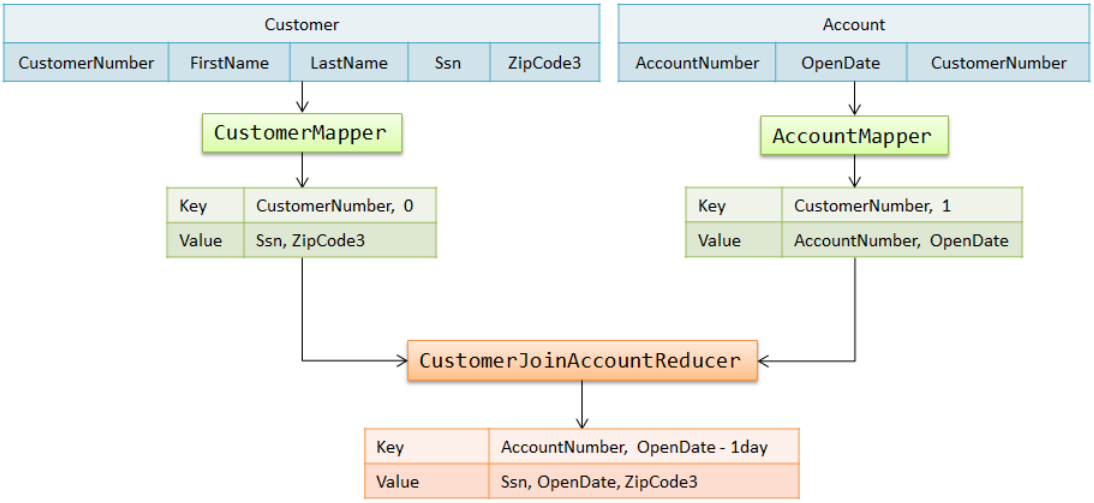

Figure 4.2: Stage 1: Customer join Account program flow diagram

```
    String[] parts = (input.toString()).split(",");

    String customerNumber = parts[COL_CUSTOMER_NUM].trim();
    String ssn = parts[COL_SSN].trim();
    String zipCode3 = parts[COL_ZIP_CODE_3].trim();

    key.set(new Text(customerNumber), ZERO);
    value = new Text(ssn + "," + zipCode3);

    output.collect(key, value);
}
```

Account Mapper

- The input is the `Account` data (formatted as a .csv file).
- The output key is `CustomerNumber` and the sort order for the key is `1`.
- The output value is `AccountNumber` and `OpenDate` of the account.

```
public void map(Object offset, Text input,
```

```
    OutputCollector<TextPair, Text> output, Reporter reporter)
    throws IOException {

    String[] parts = (input.toString()).split(",");

    String accountNumber = parts[COL_ACCOUNT_NUM].trim();
    String openDate = parts[COL_OPEN_DATE].trim();
    String customerNumber = parts[COL_CUSTOMER_NUM].trim();

    key.set(customerNumber, "1");
    value = new Text(accountNumber + "," + openDate);
    output.collect(key, value);
}
```

Customer Join Account Reducer

- The input is the sorted output from the Customer and Account mappers. Since the key value of each Customer is unique and the sort order for the Customer is 0, the Customer will be the first input value followed by a list of Accounts owned by the Customer.
- The output key is `AccountNumber` and the sort order for the key is `OpenDate`.
- The output value is
  - `Ssn`,
  - `OpenDate`, and
  - `ZipCode3`.

4.3.2 Stage 2: Strategy History Join Transaction

The second stage combines the Account and Customer data from stage 1 and the Strategy History and Transaction tables on the AccountNumber. This process requires the output from stage one and a `StrategyHistoryMapper`, `TransactionMapper` and `CustomerJoinAccountReducer` (see Figure 4.3).

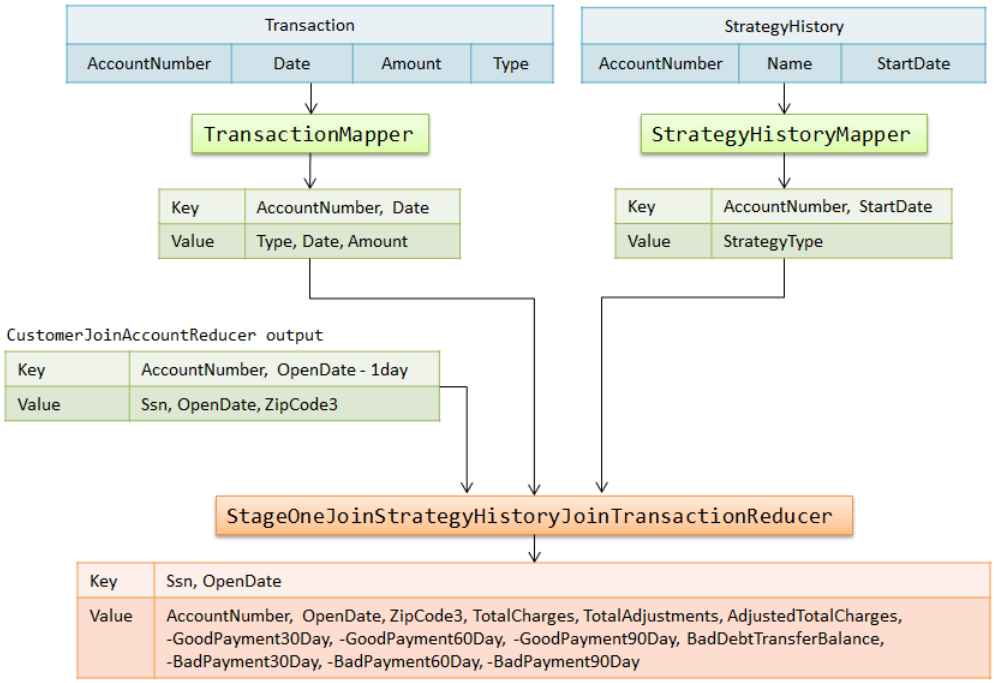

Figure 4.3: Stage 2: Stragety History join Transaction program flow diagram

Strategy History Mapper

- The input is the `Strategy History` data (formatted as a .csv file).
- The output key is `AccountNumber` and the sort order for the key is `StrategyStartDate`.
- The output value is `StrategyName` of the strategy history.

Note that multiple key/value pairs are output from this mapper. The pair for the strategy start date is output followed by a pair for each of thirty, sixty, and ninety days after the strategy start date.

```
public void map(Object offset, Text input,
    OutputCollector<TextDatePair, Text> output, Reporter reporter)
    throws IOException {

    String[] parts = (input.toString()).split(",");

    String accountNumber = parts[COL_ACCOUNT_NUMBER].trim();
    String name = parts[COL_NAME].trim();
    int type = StrategyType.getByName(name).getNumberOfDays();
```

```
    calendar.setTime(DateWritable.df.parse(parts[COL_DATE].trim()));

    key.set(accountNumber, calendar.getTime());
    value.set(Integer.toString(type));
    output.collect(key, value);

    calendar.add(Calendar.DAY_OF_MONTH, 30);
    key.set(accountNumber, calendar.getTime());
    value.set(Integer.toString(type + 1));
    output.collect(key, value);

    calendar.add(Calendar.DAY_OF_MONTH, 30);
    key.set(accountNumber, calendar.getTime());
    value.set(Integer.toString(type + 2));
    output.collect(key, value);

    calendar.add(Calendar.DAY_OF_MONTH, 30);
    key.set(accountNumber, calendar.getTime());
    value.set(Integer.toString(type + 3));
    output.collect(key, value);
}
```

Transaction Mapper

- The input is the `Transaction` data (formatted as a .csv file).
- The output key is `AccountNumber` and the sort order for the key is `TransactionDate`.
- The output value is
  - `TransactionType`,
  - `TransactionDate`, and
  - `TransactionAmount` of the transaction.

.

```
public void map(Object offset, Text input,
    OutputCollector<TextDatePair, Text> output, Reporter reporter)
    throws IOException {
```

```
    String[] parts = input.toString().split(",");

    String accountNumber = parts[COL_ACCOUNT_ID].trim();
    String date = parts[COL_DATE].trim();
    String amount = parts[COL_AMOUNT].trim();
    int type = TransactionType.getByName(parts[COL_TYPE].trim()).getOffset();

    key.set(accountNumber, date);
    value.set(type + "," + date + "," + amount);

    output.collect(key, value);
}
```

Strategy History Join Transaction Reducer

- The input is the sorted output from phase 1 and the Strategy History and Transaction mappers. Since the output from phase 1 is sorted by the OpenDate of the Account, it will always be first (because the account must be opened before any Strategy History or Transactions exist). Also, the key of each Strategy History is unique and the sort order for both the Strategy History and Transaction values are by date, so the Transaction values will be separated into each strategy time period (thirty, sixty, and ninety day) by the Strategy History values.
- The output key is `Ssn` and the sort order for the key is `OpenDate`.
- The output value is
  - `AccountNumber`,
  - `OpenDate`,
  - `ZipCode3`,
  - `TotalCharges`,
  - `TotalAdjustments`,
  - `AdjustedTotalCharges`,
  - `TotalGoodStandingPayments30Day`,

- `TotalGoodStandingPayments60Day`,
- `TotalGoodStandingPayments90Day`,
- `BadDebtTransferBalance`,
- `TotalBadDebtPayments30Day`,
- `TotalBadDebtPayments60Day`, and
- `TotalBadDebtPayments90Day`.

4.3.3 Stage 3: Combine Results

The final stage uses the output from stage 2 to gather the previous account data for each account (see Figure 4.4).

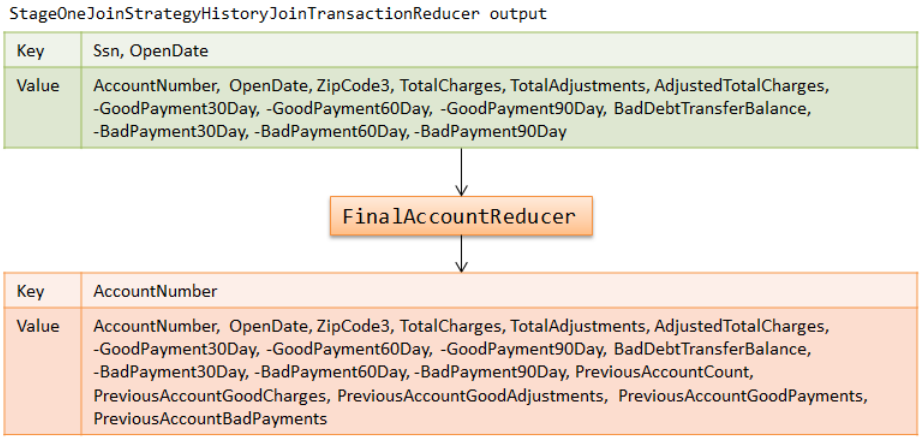

Figure 4.4: Stage 3: Combine results program flow diagram

Identity Mapper

- The input is the sorted output from stage 2.
- The output key is `AccountNumber`.

- The output value is the output value from stage 2 followed by
  - `PreviousAccountCount`,
  - `PreviousAccountGoodStandingCharges`,
  - `PreviousAccountGoodStandingAdjustments`,
  - `PreviousAccountGoodStandingPayments`, and
  - `PreviousAccountBadDebtPayments`.

4.4 Hive Solution

Because the HQL syntax is similar to the SQL syntax, we could also leverage CompanyX's existing SQL script for our Hive solution. Similar to the MySQL solution, the steps required for this implementation are:

1. Define schema for payment history analysis database.
2. Implement HQL script to load account, customer, transaction and strategy history data into database tables.
3. Transform existing SQL script to HQL for Hive execution.

In our case, the major difference between the Hive and MySQL database schema is the additional `CLUSTERED BY` parameter provided for each table definition. For each table defined in the schema, we cluster by the primary key column into 4 buckets. As in the MySQL solution, we dynamically generate the HQL load script based on a given input directory. The final load script is a series of statements of the following form– `'load data inpath <filename> into table <table name>'`. The HQL database schema can be found in Appendix C.

Chapter 5

EXPERIMENTAL RESULTS AND ANALYSIS: MYSQL, MAPREDUCE, AND HIVE PERFORMANCE COMPARISON

This chapter details the RDBMS and DDMS performance comparison experiment using our solutions to the payment history analysis case study. For each solution, we discuss the efficiency and scalability, along with the resulting implications for SMB data management. Our goal was to determine at what point Hadoop and Hive would out-perform MySQL.

The experiment was conducted on the Boise State University Onyx cluster. The cluster configuration and detailed hardware specifications are provided in Appendix B. The experiment was executed sequentially to ensure exclusive access to the server and cluster resources and to avoid the overhead of parallel task execution. Before each execution, we ensured that all nodes were alive and that the integrity of the data was unaltered. Each of the run-times reported reflects the average of three program executions.

5.1 Setup

To provide a fair and controlled environment for our benchmark analysis, we needed to ensure that the computing power of the distributed systems running Hadoop and Hive were comparable to the system running MySQL.

We installed MySQL on the master node (node00) of the Onyx cluster which is a 4-core hyper-threaded processor, yielding a total of 8 processing threads. We installed the

version 5.1.48 (x86_64) of the MySQL Community Server release for redhat-linux-gnu on the master node of Onyx. The system was configured to use 16GB for the max allowed packet size and 16MB for the net buffer length. We deleted the tables and reloaded the data for each experiment to ensure that the queries were not stored in memory.

We installed Hadoop version 0.20.2 running on Java 1.6.0_21 which includes both the HDFS and Mapreduce in both pseudo-distributed and distributed configurations. First, we configured the HDFS namenode and MapReduce JobTracker on the Onyx master node (the same as the MySQL server) along with 4 compute nodes as HDFS datanodes and MapReduce TaskTrackers– also yielding a total of 8 processing threads. Note that since the purpose of the namenode is to maintain the filesystem tree, the metadata for directories in the tree, and the datanode on which all the blocks for a given file are located [17], it does not add computing power for MapReduce or Hive in this experiment. We set the number of map tasks per job (*mapred.map.tasks*) to 8, which is 1 task per core. The maximum number of map and reduce tasks are set to 16 and 17, respectively. We left the default replication factor of 3 per block, without compression. Data for both the namenode and datanodes are stored on an `ext4` directory of the local filesystem. Then, to test Hadoop on the exact same hardware as MySQL, we configured Hadoop to run in pseudo-distributed mode on the master node. In pseudo-distributed mode, each Hadoop daemon runs in a separate Java process. After each benchmark test, we destroy and reformat the HDFS to ensure the uniform distribution of data across nodes.

We installed Hive version 0.7.0 and configured it to run on top of the same HDFS instance described above. To increase performance, we cluster the tables by primary key into buckets and store data in row format.

5.2 Execution

To determine how well each approach scales as the amount of data increases, we executed the performance benchmark on small (200MB to 500MB), medium (1GB to 5GB), and large (5GB to 10GB) data set sizes. Thus, we varied the number of accounts in the payment analysis data set from 500 to 20,000 records. Each trial was executed from a single client on the master node.

We implemented a performance benchmark test suite using bash shell scripting to help automate the execution process. The suite includes a performance benchmark executable for each of the three solutions that perform the steps enumerated below.

- **MySQL**– (1) load payment schema to MySQL database, (2) extract data from HDFS using FUSE-DFS module and load to MySQL database tables, (3) execute payment history analysis query, and (4) append query run-time to performance result file.
- **MapReduce**– (1) ensure HDFS status, (2) execute payment history analysis program, and (3) append program run-time to performance result file.
- **Hive**– (1) load payment schema to Hive database, (2) load data from HDFS to the Hive database, (3) execute HiveQL payment history analysis query, and (4) append query run-time to performance result file.

The suite also includes a script to sequentially run all of the above test scripts for each each trial size, where the experiment procedure is as follows:

1. Generate the test data set using `AccountGen` for the given trial size and record the size of the data set in the performance result file. (On first run, this also involves executing `HistogramGen` to input for `AccountGen`).
2. Execute the MapReduce performance benchmark.
3. Execute the MySQL performance benchmark.

4. Execute the Hive performance benchmark.

5.3 Results

Since we chose to generate data based on the number of accounts, we provide the approximate size of the generated data set, namely the *trial size*, for each number of accounts used in our experiments in Table 5.1. The approximate sizes are an average of generated data from each of the three trial runs.

Table 5.1: Approximate trial size of data set for number of Account records

# Accounts	Approximate trial size
500	235 MB
1000	475 MB
2500	1 GB
5000	2 GB
10000	5 GB
15000	7 GB
20000	9 GB

Figure 5.1 and Table 5.2 display the performance results of MySQL, MapReduce and Hive for each trial size with the MySQL server running on the master node of the onyx cluster and the Hadoop namenode running on the master node of the Onyx cluster and four data nodes running on four compute nodes.

Figure 5.2 and Table 5.3 display the performance results of MySQL, MapReduce and Hive for each trial size with MySQL server running on the master node of the Onyx cluster and Hadoop running in pseudo-distributed mode on the master node of the Onyx cluster.

MySQL outperforms both MapReduce and Hive for trial sizes ranging from 500 to 2,500 accounts. Since the amount of data generated for these trials is relatively small, 235MB to 1GB, the results are as expected. Aside from an anomaly at 15,000 accounts,

Table 5.2: Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running on master node plus four data nodes

# Accounts	MySQL	MapReduce	Hive
500	4.20	81.14	535.1
1000	13.83	82.55	543.64
2500	85.42	84.41	548.45
5000	392.42	83.42	553.44
10000	1518.18	88.14	557.51
15000	1390.25	86.85	581.5
20000	2367.81	88.90	582.7

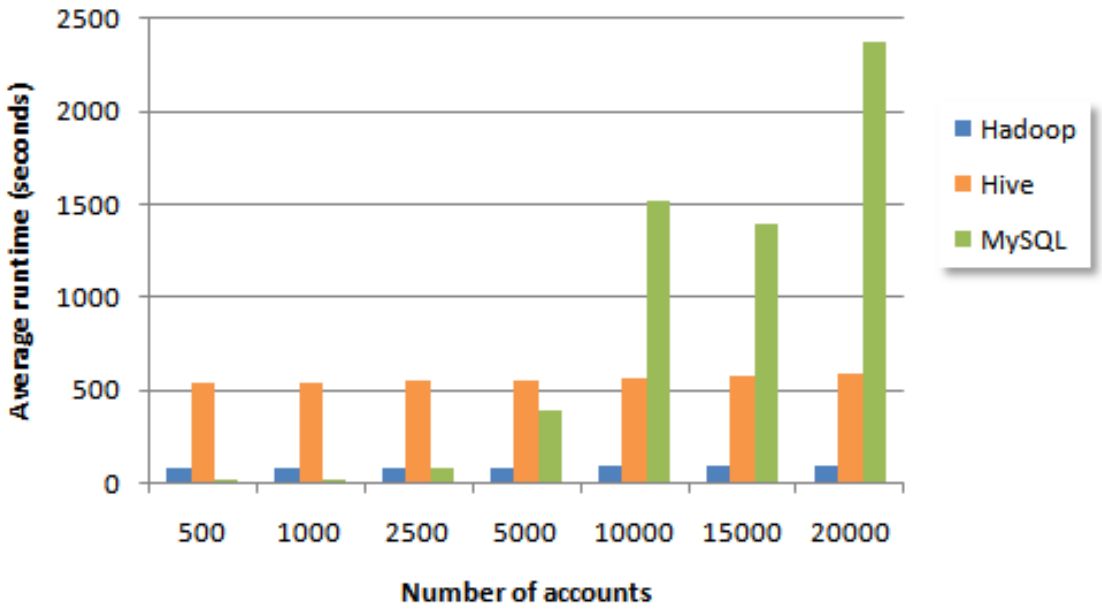

Figure 5.1: Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running on master node plus four data nodes

the run-times continue to increase as expected with the size of the data set. At 15,000 accounts the MySQL run-time decreases from 1518.18 seconds to 1390.25 seconds. The MapReduce run-time also decreases from 88.14 seconds to 86.85 seconds at this same trial size. Since the data is randomly generated and the anomaly occurred for both MySQL and Hive, it may be explained by less complex data in this data range.

Table 5.3: Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running in pseudo-distributed mode

# Accounts	MySQL	MapReduce	Hive
500	4.20	79.45	532.01
1000	13.83	79.50	531.55
2500	85.42	79.42	534.16
5000	392.42	79.44	537.00
10000	1518.18	81.48	543.51
15000	1390.25	82.39	543.95
20000	2367.81	87.47	543.64

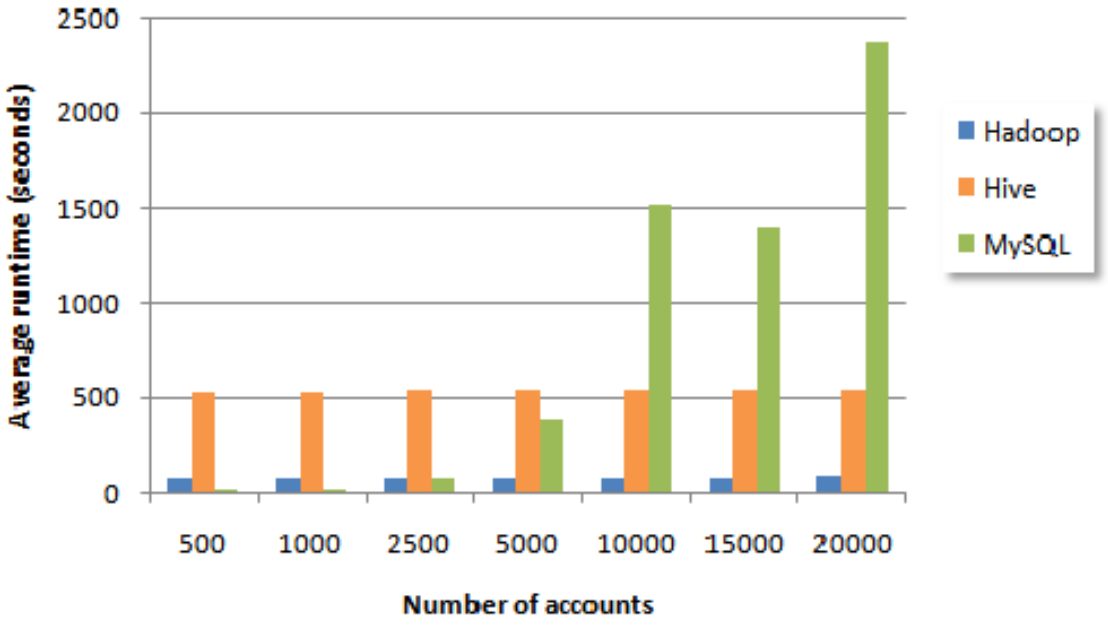

Figure 5.2: Run-time (seconds) performance of MySQL server running on master node and MapReduce / Hive running on master node in pseudo-distributed mode

In both distributed and pseudo-distributed mode, MapReduce run-times remain steady at in the range of 81 to 90 seconds for all trial sizes, converging and surpassing MySQL performance at around 2,500 accounts. Hive run-times also remain steady in the range of 535 to 583 seconds for all trials, converging and surpassing MySQL performance at

around 5,000 accounts. As expected with the relatively small trial sizes, MapReduce and Hive performed better in psuedo-distributed mode than in distributed mode with the same number of processing cores because there is no network communication and the drives are twice as fast. However, as the amount of data increases we expect better performance in distributed mode.

From these results, it is evident that MapReduce outperforms MySQL and Hive by a dramatic margin. Therefore, for test data sets greater than 1GB, MapReduce emerges as the best candidate solution for the payment history analysis case study provided by CompanyX. Furthermore, we investigate whether we can improve MapReduce performance further by varying the number of map tasks per datanode; Figure 5.3 displays the results for 1, 2, and 4 map tasks per datanode. In this case, we observe that MapReduce with 1 map task per datanode slightly outperforms 2 and 4 map tasks per datanode.

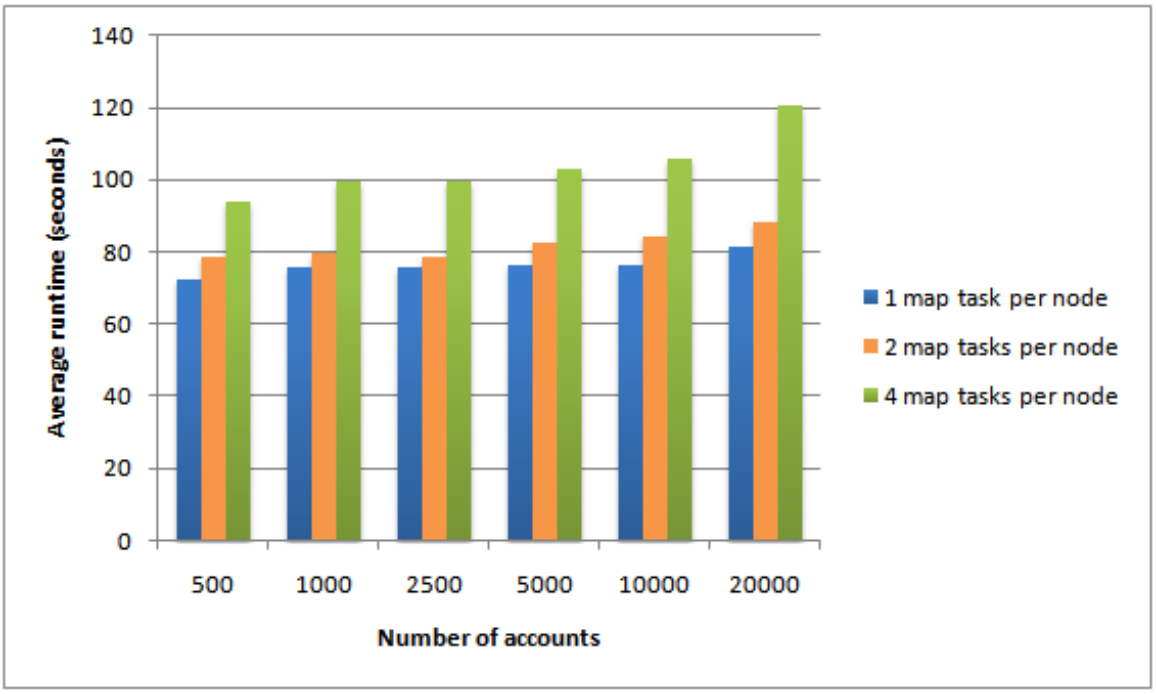

Figure 5.3: Scalability of MapReduce run-time (seconds) performance for a varied number of map tasks per datanode

Chapter 6

CONCLUSION

In this chapter we conclude by summarizing our findings, listing implications and suggesting future directions for CompanyX.

6.1 Summary

In Chapter 1 we started by introducing the concepts of this research project. We provided brief summary of BI, listed various data access and storage obstacles for SMBs, and highlighted the importance of conducting this OSS database management investigation.

In Chapter 2 we provided a background for concepts pertaining to BI storage, access, and analytical systems. Here, we outlined some major challenges faced by SMBs interested in BI analytics and discuss RDBMS and alternative DDBMS solutions. We summarized the notion of Big Data, a likely future obstacle for the SMB, and provided relevant information regarding the Hadoop solutions considered in this research: MapReduce and Hive.

In Chapter 3 we introduced the payment history analysis case study for a specific SMB. We explained how this particular SMB designs software tools to analyze current and historical payment habits on customer data and attempts forecast trends. We discussed the database management challenges faced by the SMB as their customer base expands and defined their specific case study. Here, we explained their data storage and access models, along with the requisite test data set generation utilities required to conduct the analysis.

In doing so, we formally presented the problem statement and list the inquiries we wish to address in this research project.

In Chapter 4 we defined the design and implementation for the RDBMS and DDBMS solutions. First, we discussed the detailed process of generating the test data set and provided a summarized description of the key algorithms and data structures employed by these utilities: the `HistogramGen` and `AccountGen` MapReduce programs. Second, we discussed the central components of the MySQL, Hadoop MapReduce, and Hadoop Hive solution implementation, where we explained the process by which they are used to instrument the payment history analysis case study (as presented in Chapter 3).

In Chapter 5 we detailed the RDBMS and DDMS performance comparison experiment (for the solutions of Chapter 4). Here, we discussed the efficiency and scalability for these implementations, along with the resulting implications and analysis for this SMB's data management case study (as expressed in Chapter 3). First, we discussed the software and hardware environment in which our comparative benchmark analysis is conducted. Second, we provided the procedure used to guide and execute the experiment. Finally, we presented the performance results for the various experimental trial runs: to summarize, we observed that the MapReduce solution outperforms its competitors for the case study.

6.2 Results and Implications

From the payment history analysis performance comparison between MySQL, MapReduce and Hive, we conclude that:

1. MySQL performs the best for trial sizes ranging from 500 to 2,500 accounts, but does not scale well beyond that,

2. Hive performance remained constant for all trial sizes, matching and surpassing MySQL performance at 5,000 accounts,
3. MapReduce outperformed Hive and remained constant for all trial sizes, matching and surpassing MySQL at 2,500 accounts.

Our results indicate that MapReduce is the best candidate for this case study. Therefore, we recommend that CompanyX deploy this type of distributed warehousing solution for their BI predictive analytics.

6.3 Future Direction

Beyond the scope of this project, it would be interesting to investigate RDBMS and DDMS implementations further.

- **RDBMS**
    1. Benchmark performance of several RDBMS implementations including MySQL Enterprise edition, Microsoft SQLServer, PostresQL, and Oracle.
    2. Investigate the performance ratio of parallel database solutions.
- **DDMS**
    1. MapReduce– test on several cluster environments, implement and compare performance on varied account complexity
    2. Hive– perform payment analysis benchmarks on various cluster configurations and account data

REFERENCES

[1] S. Aggarwal, L. McCabe, and A. Aggarwal. 2011 small and medium businesses routes to market study. Technical report, SMB Group, September 2011.

[2] S. W. Ambler. Relational databases 101: Looking at the whole picture. www.AgileData.org, 2009.

[3] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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

[5] Hadoop. `http://hadoop.apache.org`.

[6] Hadoop distributed file system (hdfs). `http://hadoop.apache.org/hdfs`.

[7] Hadoop MapReduce. `http://hadoop.apache.org/mapreduce`.

[8] Hive. `http://hive.apache.org`.

[9] A. Jacobs. The pathologies of Big Data. *Communications of the ACM*, 52(8):36–44, 2009.

[10] MySQL. `http://mysql.com`.

[11] D. Newberry. The role of small- and medium-sized enterprises in the futures of emerging economies. Technical report, World Research Institute, December 2006.

[12] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178. ACM, 2009.

[13] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[14] M. Stonebraker and U. Cetintemel. “One size fits all”: An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, pages 2–11. IEEE Computer Society, 2005.

[15] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[16] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD ’10, pages 1013–1020. ACM, 2010.

[17] T. White. *Hadoop: the definitive guide.* O’Reilly Media, Inc., 2011.

Appendix A

MYSQL PAYMENT HISTORY DATABASE SCHEMA

```
drop schema if exists `payment`;
CREATE SCHEMA `payment`;
use `payment`;

CREATE TABLE Customer (
    CustomerNumber varchar(10),
    FirstName      varchar(25),
    LastName       varchar(25),
    Ssn            varchar(10),
    ZipCode3       varchar(3),
    PRIMARY KEY (CustomerNumber)
);
CREATE TABLE Account (
    AccountNumber  varchar(10),
    OpenDate       date,
    CustomerNumber varchar(10),
    PRIMARY KEY (AccountNumber),
    FOREIGN KEY (CustomerNumber) REFERENCES Customer(CustomerNumber)
);
CREATE TABLE StrategyHistory (
    AccountNumber     varchar(10),
    StrategyName      varchar(15),
    StrategyStartDate date,
    PRIMARY KEY (AccountNumber, StrategyName),
    FOREIGN KEY (AccountNumber) REFERENCES Account(AccountNumber)
);
CREATE TABLE Transaction(
    AccountNumber     varchar(10),
    TransactionDate   date,
    TransactionAmount decimal(12,2),
    TransactionType   varchar(10),
    PRIMARY KEY (AccountNumber, TransactionDate,
                     TransactionAmount, TransactionType),
    FOREIGN KEY (AccountNumber) REFERENCES Account(AccountNumber)
);
```

Appendix B

CLUSTER CONFIGURATION

The benchmark experiments for MySQL, MapReduce and Hive were executed on the College of Engineering, Department of Computer Science, Onyx cluster at Boise State University. The cluster has one master node (node00) and 32 compute nodes (node01-node32), which are connected with a private Broadcom Corporation NetXtreme BCM5754 Gigabit Ethernet switch. Figure B.1 shows the layout of the Onyx cluster lab.

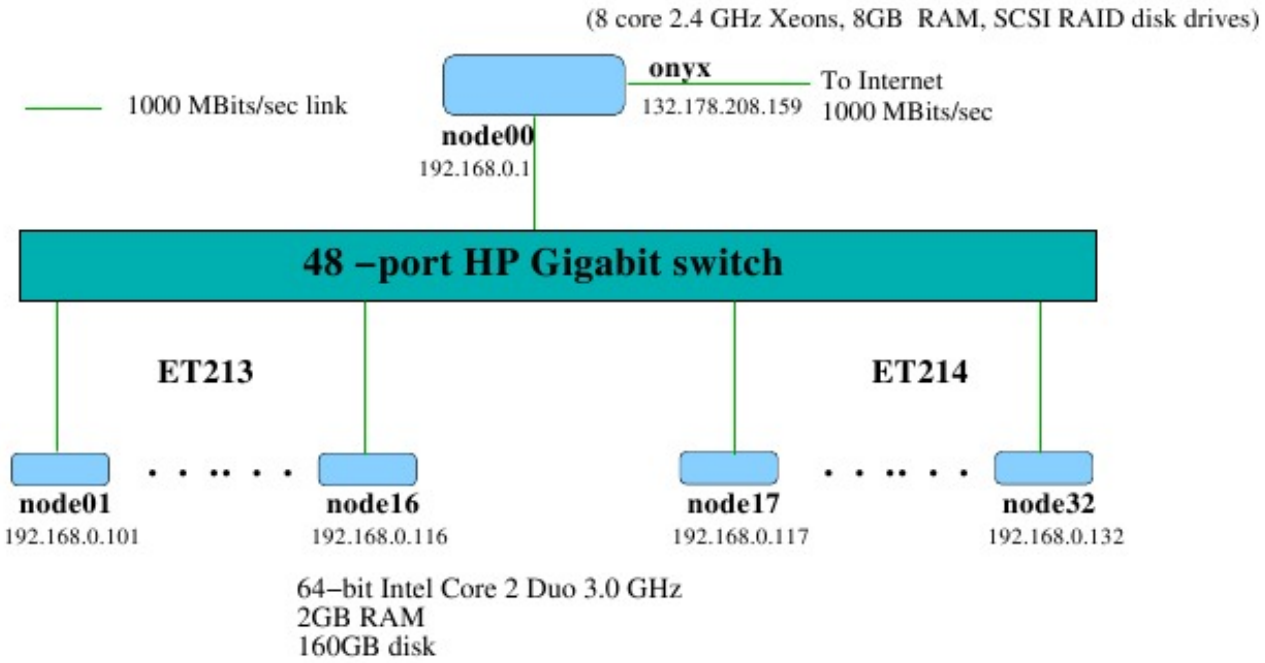

Figure B.1: Boise State University, Department of Computer Science, Onyx Cluster Lab

Master node (node00) is an Intel(R) Xeon(R) E5530 @ 2.40GHz processor with hyper-threading. It has a total of 8 processing threads (4 cores with 2 threads per core) and 15K RPM SCSI drives with RAID-6.

Each compute node (node01-node32) is an Intel(R) Core(TM)2 Duo E8400 @ 3.00GHz processor with 2 threads per node (2 cores with 1 thread per core). Each compute node has a 7200 RPM SATA disk drive.

Appendix C

HIVE PAYMENT HISTORY DATABASE SCHEMA

```
CREATE TABLE Customer (
        CustomerNumber STRING,
        FirstName STRING,
        LastName STRING,
        Ssn STRING,
        ZipCode3 STRING)
CLUSTERED BY (CustomerNumber) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';

CREATE TABLE Account (
        AccountNumber STRING,
        OpenDate STRING,
        CustomerNumber STRING)
CLUSTERED BY (AccountNumber) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';

CREATE TABLE StrategyHistory (
        AccountNumber STRING,
        StrategyName STRING,
        StrategyStartDate STRING)
CLUSTERED BY (AccountNumber) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';

CREATE TABLE Transaction (
        AccountNumber STRING,
        TransactionDate STRING,
        TransactionAmount DOUBLE,
        TransactionType STRING)
CLUSTERED BY (AccountNumber) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```