5-1-2016

Phoenix and Hive as Alternatives to RDBMS

Diana Ornelas
Boise State University

PHOENIX AND HIVE AS ALTERNATIVES TO RDBMS

by

Diana Ornelas

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2016

For my family.

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, Marissa Schmidt and Jim Conrad, as well as Dave Arnett and Vince Martino from iVinci Health 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.

ABSTRACT

There are many small and medium businesses with mid sized data sets that would like to implement low budget data management systems that will perform well with their exisiting budget and scale as more data is accumulated. One solution is to choose one of the many high-performing and cost effective Big Data management systems such as Hive and Phoenix. Another option is to use parallel database management systems whick are high-peformance alternatives but are expensive and can be complicated to implement. The purpose of this project was to compare Hive and Phoenix with MySQL to see if either are viable alternatives to relational database management systems for realtime data retrieval. The case study involved two complex stored procedures given by a local company, iVinci Health, and three simulated data sets with sizes ranging from 864.08 MB to 3.83 GB. The stored procedures take user input, generate and execute a complex query and then return the results. A web application was created to simulate how the data will be accessed in the real application. The results show that for this case study, MySQL outperforms both Phoenix and Hive. However, Hive will outperform MySQL as the data sets increase significantly in size.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API – Application Programming Interface

BI – Business Intelligence

CSS – Cascading Style Sheets

DDMS – Distributed Database Management System

HDFS – Hadoop Distribute FileSystem

HS2 – Hive Server 2

HTML – HyperText Markup Language

HiveQL – Hive Query Language

JDBC – Java Database Connectivity

JSP – Java Server Pages

RDBMS – Relational Database Management System

Chapter 1

INTRODUCTION

1.1 Overview

This is a time where there is so much data being stored for many purposes. Businesses can log every time some one visits a website, makes a payment, and many other things that they can use to give them insight on their customers. This is creating a need for new forms of Business Intelligence (BI) so businesses can have a competitive edge. Getting, storing, and analyzing digital data, as done in BI, can provide insights for organizations. The easiest part done in BI is usually storing and accessing the data since a relational database management system (RDBMS), the most popular database management system, is used to begin with most of the times. This will be a challenge as the data starts accumulating into several gigabytes or terabytes as well as when the company expands and gets more customers. That means that as the data grows, they must modify their storage methods to a parallel RDBMS or distributed among several networked systems that are managed by a distributed data management system (DDMS). Parallel RDBMSs are expensive, but there DDMSs alternatives that are open-source.

In this research, we will investigate the performance of open-source software solutions on mid-sized data sets for a small to medium business using a realistic case study. This research will extend a previous project that was done by Marissa

Hollingsworth, which we will refer to as Marissa's project from here on [10]. We will implement solutions using traditional RDBMS (MySQL) and a DDMS (Hive and Phoenix) to analyze the data. We will also create a web application for a user to input data which will be used as input to the stored procedures and return the results. Our goal is to determine if Hive or Phoenix are viable alternatives to MySQL for retrieving results in realtime.

1.2 Marissa's Project

This project builds upon a previous project done by Marissa Hollingsworth [10]. Both projects were done for a local software company, iVinci Health, that provides tools for customer information management and record analysis. This company evaluates current and historical payment habits for each customer account and attempts to predict future payment patterns based on past trends. They predict whether a payment will be on time, late, or delinquent to help with collecting payments. For example: if a customer has made late payments in the past, then the company can expect the customer to continue making payments even though they may not be on time, and avoid sending them to a collection agency [10].

A major concern for the company in the previous project was the expansion from several hundred to several thousand customers, which would increase the amount and complexity of data. This increase affects storage and access of data from their current hardware system and will force them to use a more complex system. Several risks are involved in getting a new hardware system while maintaining existing service which include: the time to implement, cost, and data integrity and service availability may be compromised as the data is being transferred. To mitigate these risks, the

company decided to compare other systems before converting their entire business architecture. Therefore the previous project addressed two of the company's major concerns: data storage model and the scalability of data analysis software [10].

Marissa was given the RDBMS solution with a small data set. She used Hadoop to generate large data sets based on statistics from the data set she was given. After generating the large data set, she implemented the solution to the same problem in Hive and MapReduce. She concluded that MapReduce was more efficient than MySQL with data sets of 1GB or larger and Hive was better with data sets larger than 5GB. The following table shows the runtime performance of MySQL running on the master node and MapReduce/Hive running on a master node plus four datanodes.

Table 1.1: Run-time (seconds) performance of MySQL server running on master node and MapReduce/Hive running on master node plus **four datanodes**

# 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

Although the project concluded that MapReduce and Hive perform better on large data sets, the company had further requirements and did not use this solution. Now they need to query the data in realtime for their new customer facing website. The problem with the previous solution is that the results for Hadoop and Hive are written to the Hadoop distributed filesystem (HDFS). However, the company needs to directly access the results instead of having to get them from HDFS first. These are the issues this project investigates.

1.3 Structure

Chapter 2 provides an overview of the technologies that were used for this project. We will give an overview of RDBMS and an overview and explanation of the Big Data technologies used: Hadoop (MapReduce, HDFS), HBase, and Phoenix.

Chapter 3 presents the case study for a specific small to medium business, iVinci Health. iVinci Health now creates patient billing and management tools for hospitals and health care providers. We discuss the challenges they face and define their specific case study. We explain their data storage model and give a brief overview of the two queries they need to accomplish. We present the problem statement and then list what this research project will try to accomplish.

In Chapter 4, we define the design and implementation for the MySQL, Phoenix, and Hive solutions. First we discuss the MySQL solution and how it is setup. Then we discuss the Phoenix solution and detail the data transformation and how the data is loaded. Next we discuss the Hive solution and how it is setup. Finally, we discuss the web application and how it connects to MySQL, Phoenix, and Hive to JDBC.

In Chapter 5, we discuss the results and analysis for this case study. Here we will go into detail about the setup for our experiment including the hardware environment and query types used to ensure thorough testing. We will also present the performance results.

Chapter 6 presents the conclusion of the research project and future work.

Chapter 2

BACKGROUND

2.1 Business Intelligence

Business Intelligence (BI) is used to help organizations make decisions. BI involves the analysis of data with the intent of enhancing business performance by helping organizations make more informed business decisions. BI technologies can be used to provide historical, current, and predictive views of business operations.

BI systems have three components: data sources, data warehousing, and analytics. Data sets can be defined as Big Data and can be sourced externally (i.e. media data, reports, etc.) or internally (account transactions, reports, etc.). A data warehouse contains a data storage that is designed to manage data gathered over time and an analytical data storage designed to manage and retrieve a historical store for predictive analysis. The last component includes the software tools used to access the analytical data store and then do the prediction procedures.

In the early development stages, businesses typically use a RDBMS for both operational and data warehousing. When the data sets are small, this strategy works well because it is cost-effective, efficient, and simple. However, as the data increases, this approach no longer works with BI analysis because of the architecture of the database [11]. This is when distributed solutions should be considered since they can be used with Big Data technologies like Phoenix and Hive. The following

sections provide background information on RBDMS, Big Data analysis technologies that are used in our research - Hadoop, Hive, HBase, Phoenix, and web application technologies.

2.2 Relational Database Management Systems

Today's most popular model is the relational database model. It is used to store and access operational data that is optimized for real-time queries on relatively small data sets. Data is organized as a set of tables with fields represented as columns and records represented as rows in the table [2]. A RBDMS is the software in charge of the storage, retrieval, security, deletion, and integrity of the data within the relational database [1]. The data can be accessed and modified in many ways through Structured Query Language (SQL) operations which are based on relational algebra.

RDBMS also have stored procedures which are operations that run within a RDBMS and can be called internally or externally [1]. They are prepared SQL code that will be saved and reused to avoid writing the same query over and over again. Stored procedures also have the ability to accept parameters and build the query according to the input data. They are especially beneficial in web applications with forms that require a POST call. For example imagine we have a web application with a form that requires a user to input a month, year, and user name. A stored procedure could take this input and perform a long and complicated query and return the results. This is useful because we can just call the stored procedure directly instead of sending the whole query over every time especially if it is a long query or is commonly used.

2.2.1 MySQL

MySQL is a freely available open-source RBDMS implementation [13]. It supports many database features including: replication, partitioning, stored procedures, views, MySQL Connectors used to build applications in multiple languages, and MySQL Workbench, which is a visual tool used for modeling, SQL development, and SQL administration [13].

A common MySQL deployment will include a server that is installed on one high-end server that accepts local and remote client queries. The database is limited to the hard drives of the servers so when the amount of data exceeds the storage capacity, then the model will fail and a DBMS with an underlying distributed storage will need to be deployed.

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 [18]. 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.

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 [14]. 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 [17]. In contrast, Hadoop 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 small to medium businesses 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 [14].

Figure 2.1 shows a high level architecture for MySQL, Hive and Phoenix.

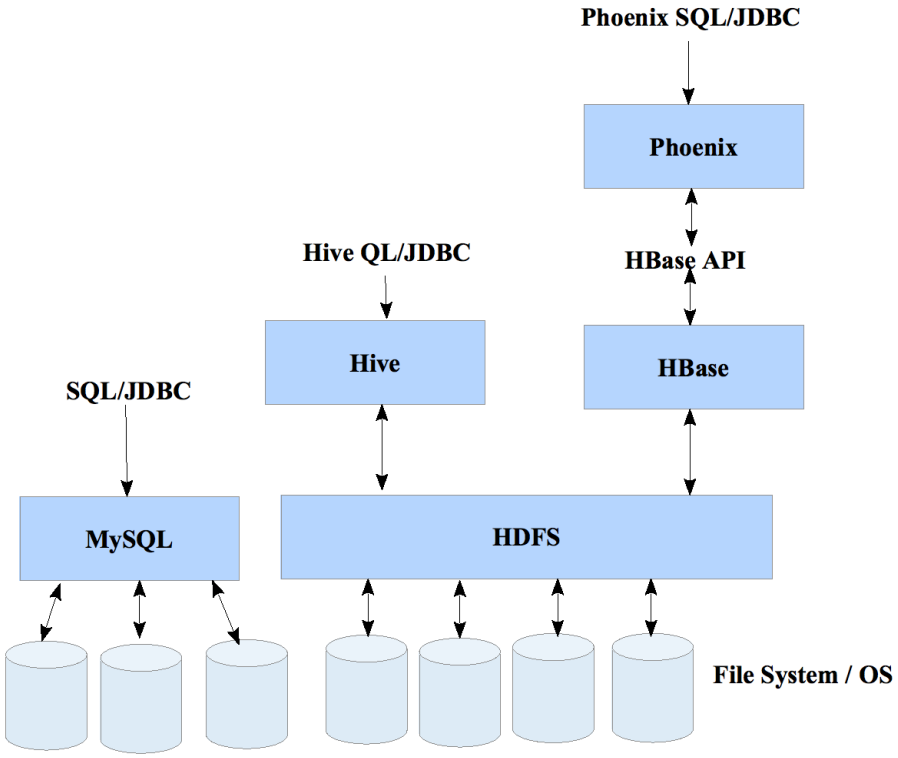

Figure 2.1: `Architecture for MySQL, Hive, and Phoenix`

2.3.1 Hadoop Distributed Filesystem

Hadoop is an open-source implementation of the MapReduce framework developed by Google. Hadoop provides several open-source projects for reliable, scalable, and distributed computing [5]. Our project will use the Hadoop Distributed Filesystem (HDFS) [6], Phoenix [15], and Hive [9].

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. It was developed as an abstraction of the *map* and *reduce* primitives present in many functional languages [3, 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

Implementing MapReduce jobs may take a few hours or even a few days, which is why Facebook decided to develop Hive [19]. Hive is a layer on top of Hadoop that allows the user to query and manage data using Hive Query Language (HiveQL). Hive converts the queries into MapReduce jobs and HDFS operations with several optimizations and then executes it on the Hadoop cluster [9]. It is convenient to use because HiveQL is very similar to SQL, which is commonly used by in database systems.

2.3.4 HBase

HBase is a NoSQL column-oriented database system that is distributed, persistent, consistent, and is built on top of HDFS [4]. HBase lets us store and process large amounts of data ac ross multiple machines. The benefits of using HBase are that

it is cost effective, dependable, and is fast at retrieving data where it can give us a random read/write access in realtime [8].

HBase has a basic client API that includes two commonly used features: scans and filters. Scan is used to get data from HBase. It allows us to iterate through a range of rows and lets you limit which columns are returned. Filters are more powerful and effective for working with data. Filters are used to select certain columns or cells based on a variety of conditions. There are many predefined filters provided by HBase but it allows custom filters implementations.

HBase uses regions as the basic unit of scalability and load balancing. Contiguous ranges of rows are stored together. Initially there is one region for a table and as it becomes too large, it splits into two and so on. These regions are served by one region server. A region server can have various regions and are in charge of handling all requests for their regions (to add, remove, etc.). HBase also has a master server that is in charge of assigning regions to the region servers for load balancing.

HBase has a coprocessor framework that provides a way to run custom code on a region server. Coprocessors allow us to execute code on a per-region basis, giving trigger-like functionality similar to that of stored procedures. HBase has some classes based on the coprocessor framework which fall into two main groups: Observers and Endpoints. Observers have callback functions (hooks) that are executed when certain events occur, including user-generated, server-internal, and automated events. Endpoints resemble stored procedures where we can invoke them anytime from the client. The endpoint implementation will then be executed at the targeted region or regions and the results from that execution will be returned to the client [4].

2.3.5 Phoenix

While HBase provides a good framework for realtime random access read/write, there is no well-defined API and comes with no built-in coprocessors. Apache Phoenix's mission is to become the standard means of data access through HBase with the use of a well-defined, industry standard API. It is used by a few big companies for data analytics including: ebay, Hortonworks, Dell, Intel, and Alibaba.com [15].

Apache Phoenix is an open source, parallel and relational database layer that is built on top of HBase. It connects to a client-embedded JDBC driver to enable users to create, delete, upsert, and query data, and to provide additional functionality used by most SQL languages. They use their own language that is very similar to SQL. Phoenix compiles the queries and statements from the client into a series of HBase scans, filters, and coprocessors and then runs them to produce a result set that is retrieved in the order of milliseconds for smaller queries and seconds when using millions of rows [4].

2.4 Web Interface

Web applications are widely used by businesses to communicate and interact with potential customers. A web application is a program running on a server that responds to client requests to retrieve and submit data through the backend. They can be deployed at a relatively low cost and typically don't require the customer to install anything other than a web browser. Our web application is built using: HyperText Markup Language (HTML), Cascading Style Sheet (CSS), and JavaServer Pages (JSP).

2.4.1 HTML

HTML is used for writing web pages and is the building block for all web interfaces [16]. HTML is written using HTML elements that consist of tags enclosed in angle brackets. It describes the contents of the page such as paragraphs, tables, headings, images, links, and lists.

2.4.2 CSS

CSS describes the way the contents in HTML should look such as fonts, colors, and alignment [16]. The web originally did not use CSS and instead used HTML formatting tags to indicate if the text should be a certain color, font, etc., but this defeats the purpose of HTML. It lets the developer clearly distinguish between the content and the way the content will look in a web browser.

2.4.3 JSP

JSP is a technology based on the Java language that is used to create dynamic, platform-independent web applications [12]. JSP files are basically HTML files that allow for special tags where we can add Java code to provide dynamic content. One of the benefits of JSP is that it has access to the Java APIs, including the JDBC API that can be used to connect to databases.

2.4.4 JDBC

JDBC is a Java database connectivity driver provided by Oracle [12]. JDBC is an Application Programming Interface (API) for the Java programming language that defines how a client may access a database.

Chapter 3

DATA RETRIEVAL ANALYSIS: A CASE STUDY AND PROBLEM STATEMENT

This case study was provided by a local software company, iVinci Health. This company specializes in patient billing and management tools for hospitals and healthcare providers. They provide a website used by hospitals for finances and billing of health services for their patients. The website shows patient billing, management tools, and financial options. This case study will focus on two of the tools. The first one gets user input including a start date, end date, and other input used to do data analysis that returns the number of accounts that paid, the payment agency, and the total amount paid for every month between the start and end date. The second chart gets user input to do data analysis that gives a payment summary that goes back up to three years. This case study will strictly focus on getting accurate results and testing the retrieval speeds to see if there is a viable alternative to RDBMS solution.

This company keeps expanding and attaining more customers which means they have more data for the data analysis. This was the concern investigated in Marissas project that used Hadoop and Hive to show they are better for doing the predictive data analysis on large data sets. However, the new challenge is that the results were stored in HDFS and there is no direct way to get the results from HDFS. They have made several changes since Marissa's project including the website for their

customers. The new challenge that iVinci Health wants to address is whether there is a viable alternative to the RDBMS solution for real-time data retrieval using a web interface. The company provided three simulated data sets that were to be used as long as everything is secure (a Non-Disclosure Agreement was signed). The data provided was not actual client data. It was produced and constructed to be realistic and representative of the field structure and data typically used in hospital billing systems.

3.1 The Case Study

3.1.1 Data Storage and Relationship Model

The company uses RBDMS for data management. The data includes the following tables: *RptAllOperationsBatchCollectionsReportSummary*, *RptAllOperationsBatchCollectionsReportPaymentSummary*, and *RptCashCollectionsMonthlyReportSummary*

RptAllOperationsBatchCollectionsReportSummary: The primary key is the *RptAllOperationsBatchCollectionsReportSummaryID* attribute. Each *RptAllOperationsBatchCollectionsReportSummary* entity stores 75 columns including *batchYearMonth*, *batchFiscalYear*, *location*, as well as many others.

RptAllOperationsBatchCollectionsReportPaymentSummary: This entity has a foreign key, *RptAllOperationsBatchCollectionsReportSummaryID*, from *RptAllOperationsBatchCollectionsReportSummary*. Each *RptAllOperationsBatchCollectionsReportPaymentSummary* entity stores 79 columns.

RptCashCollectionsMonthlyReportSummary: This entity does not have a primary or foreign key. It stores 72 columns including *insurancePlans, paymentSource*, *location*, *accounts*, and *transactionYearMonth*.

3.1.2 Data Retrieval

iVinci Health uses stored procedures to retrieve the data stored in the tables. These stored procedures accept input parameters that are used to create and run a query. Data retrieval is done though join, union, and other SQL queries. The input parameters to the stored procedures are inserted from the web application form. The results are returned as a table on the web application. There are two stored procedures used for this case study:

1. `spRptCashCollectionsMonthlyReport`: retrieves data from the *RptCashCollectionsMonthlyReportSummary* table.
2. `spRptAllOperationsBatchCollection`: retrieves data from *RptAllOperationsBatchCollectionsReportSummary* and *RptAllOperationsBatchCollectionsReportPaymentSummary* tables.

The *spRptCashCollectionsMonthlyReport* stored procedure takes in seven input parameters: *pivotBy*, *viewPaymentsBy*, *billingApplicationID*, *filter*, *startYearMonth*, *endYearMonth*, and *viewDatesBy*. The *filter* input parameter is XML with 13 elements. Each element has a key and a value where they key is the column name of the table and value is a list of values for that column. The value will contain "All" if a particular column will not be filtered.

The spRptAllOperationsBatchCollection stored procedure takes in four input parameters: *billingApplicationID*, *filter*, *paymentType*, and *excludeCash*. The *filter* input parameter is XML with 24 elements. Each element has a key and a value where the key is the column name of the table and value is a list of values for that column or "All" if there will not be a filter for that particular column.

Figure 3.1: Generalized query generated by the *spRptCashCollectionsMonthlyReport* stored procedure

```
SELECT substring( datename( mm, convert( datetime, convert(
    varchar(8), TransactionYearMonth) + '01', 112)), 0, 4) + '
    -' + substring( convert( varchar(8), TransactionYearMonth)
    , 3, 2)) as val1, TransactionYearMonth as val2,
    other_fields
FROM RptCashCollectionsMonthlyReportSummary pf
WHERE input_parameters_conditions
GROUP BY TransactionDate, PaymentAgencyName
UNION
SELECT substring( datename(mm, convert( datetime, convert(
    varchar(8), TransactionYearMonth) + '01', 112)), 0, 4)+ '-
    ' +substring( convert( varchar(8), TransactionYearMonth),
    3, 2)) as val1,TransactionYearMonth as  val2, other_fields
FROM RptCashCollectionsMonthlyReportSummary pf
WHERE input_parameters_conditions
GROUP BY TransactionDate
ORDER BY val1, val2 asc
```

Figure 3.1 shows that the query generated by the stored procedure is the union of two queries. Both of the queries are similar, the differences are that the first query groups by one more column, *PaymentAgencyName*, and the conditions can vary more based on user input. The query results will be sorted in ascending order based on the *zValue* and *otherValue* fields, which are values of a particular column in the *RptCashCollectionsMonthlyReportSummary* table.

Figure 3.2 shows that the second generated query consists of a nested query, right join, order by, and other functions. The inner query retrieves values from the *RptAllOperationsBatchCollectionsReportSummary* table based on some of the input parameters that include: *startDate*, *endDate*, and *paymentSource*. Results will be returned for all months between the start and end date and will do this for each

Figure 3.2: Generalized query generated by the *spRptAllOperationsBatchCollection* stored procedure

```
SELECT substring( datename (mm, convert( datetime, convert(
    varchar, taobcp.YearMonth) + '15')), 0, 4) + '-' +
    substring (convert (varchar, taobp.YearMonth), 3, 2) as
    batch, ..., sum( case when convert( int, replace(aobcp.
    Months,'+','')) = 1 then aobcp.TransactionAmount else 0
    end) paymentMonth1, ....
FROM (
    SELECT some_fields
    FROM RptAllOperationsBatchCollectionsReportPaymentSummary
    WHERE input_parameters_conditions
) aobcp
RIGHT JOIN (
    SELECT some_fields
    FROM RptAllOperationsBatchCollectionsReportSummary aobc
    WHERE input_parameters_conditions
    GROUP BY Year, Month
) AS taobcp
ON taobcp.FiscalYear = aobcp.FiscalYear and taobcp.YearMonth
    = aobcp.YearMonth
GROUP BY taobcp.FiscalYear, taobcp.YearMonth, aobcp.
    PaymentSource, taobcp.Date, taobcp.Listed, taobcp.
    adjustedFace, taobcp.accountCount
ORDER BY taobcp.YearMonth, aobcp.PaymentSource
```

payment source that can include patient cash, payor cash, or charity.

There is a right join with *RptAllOperationsBatchCollectionsReportPaymentSummary* which joins the results from two similar queries. The right join takes place when the fiscal year and fiscal month on both tables are the same.

There is a group by with seven different fields: *fiscalYear*, *fiscalMonth*, *paymentSource*, *batchDate*, *listed*, *adjustedFace* and *accountCount*. The results are sorted in ascending order based on two columns: *batchYearMonth* and *paymentSource*.

The select statement of the outer query formats the value of a date, calculates how much has been paid in the first 12 months, 18 months, 24 months, 36 months, and calculates the liquidation. It uses *substring*, *datename*, and *convert* to format the date. For example, if the *yearMonth* is 201202 (February 2012) then it will format it to print the first three letters of the month followed by the last two digits of the year: Feb 12.

3.2 Problem Statement

iVinci Health is faced with challenges pertaining to Big Data:

1. They want direct real-time access to data through stored procedures (or something similar).
2. The database access times increase drastically as the amount and complexity of data accumulates over time.

To investigate solutions to these challenges we want to know if we can use a distributed model, such as Hive or Phoenix, as viable alternatives to RDBMS stored procedures to access data. We know Hive performs well for Big Data analysis but how will it perform in this particular case? Thus, the goals of the research project are to:

- Design and implement a web application that will directly access data based on user input.
- Implement and compare a Phoenix solution.
- Implement and compare a Hive solution.

Therefore, we will address the following questions:

1. Is there a viable alternative that will return results close to realtime and outperform RDBMS?

2. Will the existing data schema work with each solution?
3. How much cost and effort will it take to deploy each solution?

Chapter 4

DESIGN AND IMPLEMENTATION

The design and implementation process of this case study will be split into four parts: MySQL solution, Phoenix solution, Hive solution, and web application implementation. First, we will discuss the implementation of the MySQL solution, then the Phoenix solution, followed by the Hive solution. Finally we will discuss the web application implementation that connects the user interface to MySQL, Phoenix, and Hive solutions respectively to retrieve the results.

4.1 MySQL Solution

iVinci Health already has a RDBMS implementation for the data retrieval for this case study using SQL server which was an advantage in terms of translating the schema and stored procedures into a MySQL solution. We developed this solution with the following steps:

1. Define schema for *RptCashCollectionsMonthlyReportSummary*, textitRptAllOperationsBatchCollectionsReportPaymentSummary, and textitRptAllOperations-BatchCollectionsReportSummary tables.
2. Implement MySQL script to load data into the tables.
3. Update existing SQL stored procedures to execute in MySQL.

iVinci Health supplied the database schema and stored procedures for their SQL solution, which were easily converted to MySQL with slight syntax modifications. They also provided three simulated data sets, so a script was created to load these data sets into the appropriate tables. The load script is a series of statements in the following form: `'load data local infile <filename> into table <table name>'` where we specified what the fields are terminated by, what the lines are terminated by, and to ignore the first line since it contains the names of the columns.

4.2 Phoenix Solution

iVinci's SQL solution was also similar to the solution implemented for Phoenix. The solution was developed in the following steps:

1. Define the schema for the three tables.
2. Implement script to load the data sets into the appropriate tables.
3. Translate the SQL stored procedure to a Java program that is ran by JSP.

Transforming the Data: The data given by iVinci Health had to be changed. The data given by iVinci was separated by the '|' character and we had to change it to be separated by commas since csv files are used for bulk loading. We had to add a column to the *RptCashCollectionsMonthlyReportSummary* table to be used as the rowkey since this table doesn't have a primary key that Phoenix supports. Phoenix does not offer the option to auto-increment a field so we had to generate the auto-increment rowkey values before loading the data into the tables.

Schema and Loading Data into Tables: The schema for the Phoenix database was derived from the schema given by iVinci Health with minimal syntax

changes and adding the key column discussed above. Loading the tables is very different than the MySQL solution. Phoenix provides two ways to load the data into the tables: bulk load using MapReduce or a single-threaded *psql* python script. To save time, MapReduce was chosen because it is distributed, however, since there is only one reducer it was relatively slow even though it did allow for multiple mappers. A Phoenix client jar file was used to execute the CsvBulkLoadTool MapReduce job. The Hadoop job was executed using a command similar to the following: `'hadoop jar <phoenix client jar> org.apache.phoenix.mapreduce.CsvBulkLoadTool --table <table name> --input <files to load>'`.

Translating SQL Stored Procedures : Since Phoenix does not offer stored procedures, we implemented an equivalent solution in Java since JSP is used for the web application. The Java solution accepts input parameters and generates a query that will be executed and return the results.

4.3 Hive Solution

Hive Query Language (HiveQL) is very similar to SQL, so we based our Hive solution on iVinci's SQL solution. This solution was very similar to MySQL and the steps taken for this implementation were:

1. Define schema for *RptCashCollectionsMonthlyReportSummary*, textitRptAllOperationsBatchCollectionsReportPaymentSummary, and textitRptAllOperations-BatchCollectionsReportSummary tables.
2. Implement HQL script to load data into tables.
3. Translate the SQL stored procedures to a Java equivalent program ran by JSP.

The data did not have to be transformed for the Hive solution. However, there were some changes that had to be made in the schema to avoid transforming the data. At the end of the defined schema we specified that the fields where terminated by `'|' instead of specifying this when we loaded the data. To load the data we used` data inpath ¡files to load¿ into table ¡table name¿;'

4.4 Web Application Implementation

Now that we described the solutions for this case study we need a way to have direct access to the results, which is through a web application. JSP was chosen for the home page and the results page because Phoenix already connects through JDBC and Hive comes with libraries to connect to JDBC. Figure 4.1 models the flow of the web application:

4.4.1 Home Page

The home page is a JSP page that contains a form with all the required and optional fields used in the stored procedures or Java equivalent program to generate the query to be executed. There are six home pages: one for each solution and one for each stored procedure or stored procedure equivalent. In the following code snippet we can see how the form looks for the MySQL spRptAllOperationsBatchCollection stored procedure home page.

```
<html>
    <head>
        <title>All Operations </title>
        <link rel="stylesheet" type="text/css" href="index.css"/>
    </head>
```

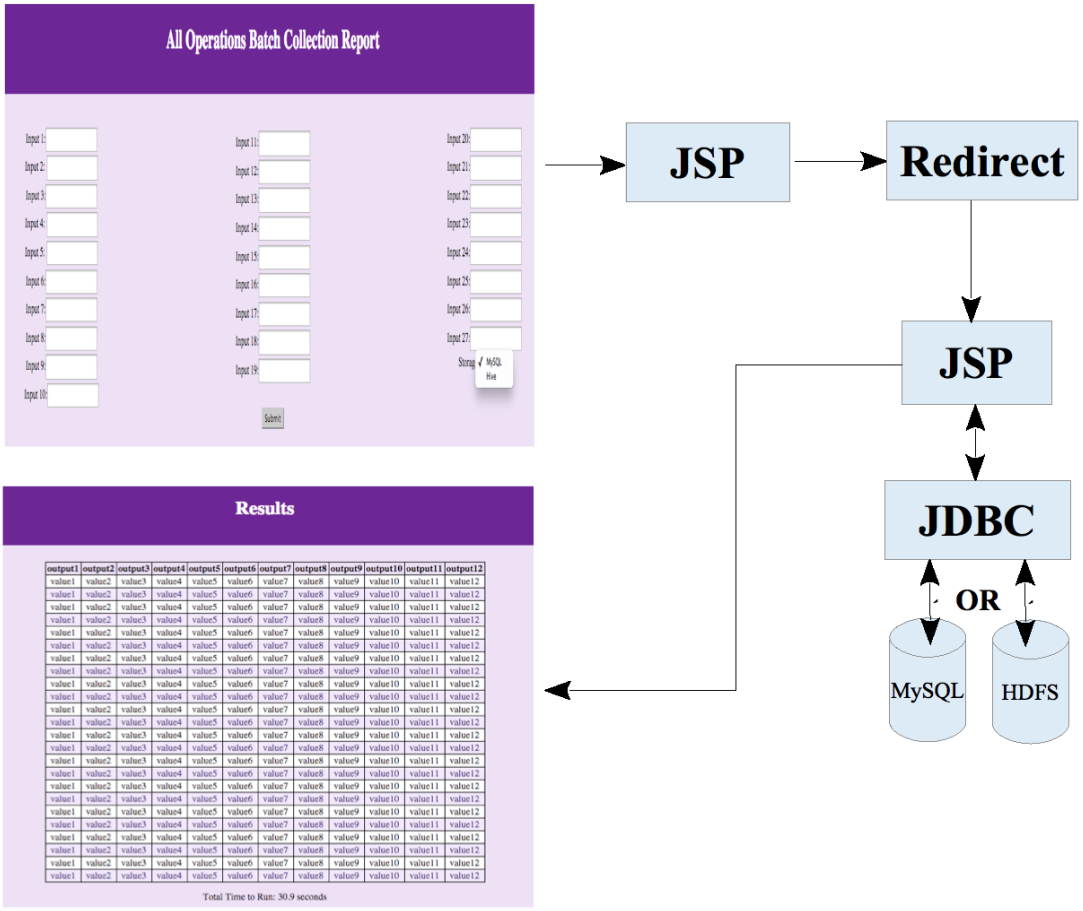

Figure 4.1: JSP flow diagram

```
<body class="body">
    <div id="header">
        <h1>All Operations Batch Collection Report</h1>
    </div>
    <form method="post" action="AllOpsMySQLResults.jsp">
        Billing Application ID:
        <input type="text" name="billingapplicationid"/>
        Payment Type:
        <input type="text" name="paymenttype"/>
        Exclude Cash:
        <input type="text" name="excludecash"/>
        <h1>Filters:</h1>
        Initial Admit Dept ID:
        <input name="initialadmitdeptid" type="text"/>
        Primary Payor ID:
```

```
            <input name="primarypayorid" type="text"/>
 .....
            Next Statement Within Days ID:
            <input name="nextstatementwithindaysid" type="text"/>
            <input type="submit" value="Submit" name="indexSubmit">
        </form>
    </body>
</html>
```

4.4.2 Results Page

From the previous code snippet we can see that after the user clicks the submit button it goes to the results JSP page. The results page will show a formatted table with the results or it will display a message indicating that no results were found. There are six JSP result pages: one for each stored procedure or Java equivalent program and for each technology. The following code snippet shows the results JSP file:

```
<html>
    <head>
        <link rel="stylesheet" type="text/css" href="index.css"/>
        <title>Cash Collections Results</title>
    </head>
    <body>
        <%
            //Connect to JDBC
            AllOperations ao = new AllOperations(request);
            String table = ao.runQuery("mysql", JDBCConnection);
        %>
        <%= table %>
    </body>
</html>
```

The code snippet shows it uses JDBC to connect to the appropriate database and how an AllOperation object is created and how it calls the run query method. The run query method takes in two parameters: one String that specifies what technology

to use and the JDBC connection so it can run and retrieve the query results using this connection. One parameter in the home page form requires the user to enter the query type: user input, all, half, and small. The user input takes the input from the user and the others will call a method to generate the query with already defined input values.

4.4.3 Using JDBC

Connecting to JDBC using MySQL was simple. We can connect using a statement similar to the following:

```
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManager.getConnection("jdbc:mysql://host:port/db",
"username", "password");
...
conn.close();
```

Connecting to JDBC using Phoenix was also simple since Phoenix was designed so results could be accessed using JDBC. The HBase client and Phoenix client jars need to be in the build path. We can connect to JDBC by using code similar to:

```
    Class.forName("org.apache.phoenix.jdbc.PhoenixDriver");
    Properties prop = new Properties();
    Connection conn =  DriverManager.getConnection("jdbc:phoenix[:zk_quorum]
[:zk_port][:zk_hbase_path][:headless_keytab_file:principal]");
    ....
    conn.close();
```

Connecting JDBC to Hive was the most complicated one. We have to have a few jar files in the build path: hive-exec, hive-jdbc, and hive-service. Before connecting to JDBC, HiveServer2 (HS2) and beeline must be started since this will be the client

JDBC will connect to. HS2 is a server interface that enables clients to execute queries and return the results. Beeline is a JDBC client that depends on HS2. To use JDBC we had to use statements similar to the following:

```
Class.forName("org.apache.hive.jdbc.HiveDriver");
Connection conn = DriverManager.getConnection("jdbc:hive2://<host>:<port>",
"<user>", "<password>");
...
conn.close();
```

Note from the previous code we must use the host and port where beeline is running.

Chapter 5

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter will outline RDBMS and DDMS performance comparison for our case study using our solutions from Chapter 4. For each solution we will discuss the efficiency and use the results to determine if Phoenix or Hive are viable alternatives to RDBMS with results retrieved in realtime.

The experiment was conducted on the Boise State University Onyx cluster. The cluster configurations and hardware specifications are in Appendix A. The experiments were executed one at a time to ensure we had exclusive access to the server and cluster resources. Before each execution we also made sure all the nodes were up and made sure the nodes were not being used by another user to ensure exclusive access to resources. Each average is calculated from an average of two program executions.

5.1 Setup

MySQL was installed on the master node of the onyx cluster which is a 6-core hyper-threaded processor with a total of 24 processing threads. We installed the 10.0.20-MariaDB (x86_64) version of the MariaDB Server release for Linux on the master node of Onyx. The tables were deleted and reloaded for each experiment to ensure that queries were not stored in memory.

We decided to install Hadoop 1.2.1 after seeing that Hadoop 2.4.1 used too many resources for Onyx and gave runtimes up to 50% more than Hadoop 1.2.1. Hadoop is setup to run Java 1.8.0-65. We had to configure the HDFS *namenode* and MapReduce JobTracker on the Onyx master node along with four other nodes that are the HDFS *datanodes* and MapReduce TaskTrackers which used 8 processing threads. Having the master node as the *namenode* does not really add any more computing power for Hive or Phoenix. The *namenode* is in charge of the directory tree of all the files in the filesystem and tracks where the data is stored across the cluster [20] so it does not provide any additional computing power for Hive or HBase in this experiment. For the four *datanodes* we set the number of map tasks per job to be eight, one for each core, and set the maximum number of map and reduce task to be set to 16 and 17, respectively. We also setup the cluster for two and four *datanodes* where two nodes was set to have a maximum of four map tasks per job and eight reduce tasks per job and eight nodes was set to have a maximum of 16 map tasks per job and 17 reduce tasks per job. Data for the *namenode* and *datanodes* are stored on a `ext4` directory of the local filesystem. To ensure that the data was distributed evenly across the *datanodes*, we reformatted HDFS after each benchmark test.

We installed version 1.0.0 of Hive and configured it to run on top of the HDFS specified above.

We installed version 0.94.26 of HBase and configured it to run on top of the HDFS specified above. HBase uses the HDFS *namenode* (onyx master) as the HMaster, the HDFS *datanodes* as the regionservers, and the zookeeper quorum consists of only the onyx master since the cluster was small. Having the master node as the zookeeper quorum does not really add any more computing power to HBase or Phoenix. The zookeeper quorum is a list of servers used by zookeeper, which is a centralized service

to enable synchronization across a cluster [4]. HBase was configured to use the built-in zookeeper by setting HBASE_MANAGES_ZK to true in the hbase-env.sh file.

We installed version 3.3.1 and 3.1.0 of Phoenix and configured it to run on top of HBase. Two versions were necessary since the all operations method works only with version 3.3.1 since this is when inner joins became available and the cash collection method only works with version 3.1.0. Phoenix 3.3.1 causes cash collections to time out after 10 minutes due to not having enough resources, but it worked on Phoenix 3.1.0.

5.2 Execution

iVinci Health provided simulated three data sets for each table. The size for these data sets range from 864.08 MB to 3.83 GB. The following table shows the size, number of rows, and number of columns for each table. Notice we shorted the tables names. *RptAllOperationsBatchCollectionsReportSummary* was shortened to *Account*, *RptAllOperationsBatchCollectionsReportPaymentSummary* was shortened to *Payment*, and *RptCashCollectionsMonthlyReportSummary* was shortened to *Cash Collections*.

Table 5.1: Data Set Information

Table Name	Size	Rows	Columns
Account	864.08 MB	1,444,417	75
Payment	1.07 GB	1,711,465	79
Cash Collections	3.83 GB	5,787,276	72

We had to load the data before running any tests. The loading times for each technology varied. Phoenix was the slowest followed by MySQL and Hive. Hive was the fastest and was able to load from HDFS in matter of seconds. MySQL took a

few minutes: loading the *Account* table took about 3.5 minutes, *Payment* table took about 2 minutes, and *Cash Collections* table took about seven minutes. Phoenix took several minutes and the number of nodes in the cluster didn't make a difference since there was only one reducer. Phoenix took from 28-35 minutes to load the *Cash Collections* table, 6-8 minutes to load the *Account* table, and 7-9 minutes to load the *Payment* table. The times to load won't be much of an issue since this will in theory only happen once.

We tested each stored procedure solution by generating three queries with different filters. One uses filters to do analysis on all data by setting filters to all possible values, the second uses XML filters to do analysis on about half of the data by setting the XML filters to be half of all possible values, and the last one uses filters to analyze a small amount of data by setting XML filters to use a small number of all possible values. We will refer to them as all, half, and small.

We implemented three scripts to help automate the execution process for each solution: Hive, Phoenix, and MySQL. These scripts perform the steps below, in order.

- **MySQL**– (1) load schema to MySQL database, (2) load given data to MySQL database tables, (3) create the stored procedures (4) execute and time the queries and append the result to a file.
- **Hive**– (1) create and reformat the Hadoop cluster and ensure HDFS status, (2) load the data sets to HDFS and create the Hive tables, and (3) execute and time the queries and append the result to a file.
- **Phoenix**– (1) create and reformat the Hadoop cluster and ensure HDFS status, (2) load the data sets to HDFS, (3) run HBase on top of HDFS, (4) create and load Phoenix tables, and (5) execute and time the queries and append the result

to a file.

5.3 Results

We used the sizes specified in Table 5.1 of the previous section. We also designed three unique queries where one query used the XML filters to get all the data, another used XML filters to retrieve approximately half of all the possible values, and the last used XML filters to select a small number of possible values. Additionally, the All Operations stored procedure equivalent for Phoenix ran on version 3.3.1 and the Cash Collections stored procedure equivalent for Phoenix ran on version 3.1.0.

Table 5.2 shows the performance results of MySQL, Phoenix, and Hive for the three different query types and the two stored procedures. MySQL ran on the master node while Phoenix and Hive ran on the master node of onyx and two *datanodes*. The Hive queries were configured to use four MapReduce jobs for Cash Collections and five MapReduce jobs for All Operations.

Table 5.2: Run-time (seconds) performance of MySQL server running on master node and Phoenix and Hive running on master node plus **2 datanodes**

Stored Procedure	Query Type	MySQL	Phoenix	Hive
All Operations	All	47.35	Time Out	87.29
All Operations	Half	14.34	60.27	85.79
AllOperations	Small	20.59	60.6	85.50
Cash Collections	All	58.74	361.59	92.51
Cash Collections	Half	31.25	195.61	89.61
Cash Collections	Small	31.24	197.72	92.02

Table 5.3 shows the performance results of the same configuration but with double the *datanodes* for a total of four.

Table 5.3: Run-time (seconds) performance of MySQL server running on master node and Phoenix and Hive running on master node plus **4 datanodes**

Stored Procedure	Query Type	MySQL	Phoenix	Hive
All Operations	All	47.35	Timed Out	58.54
All Operations	Half	14.34	64.11	62.10
AllOperations	Small	20.59	57.56	58.06
Cash Collections	All	58.74	327.65	85.09
Cash Collections	Half	31.25	190.03	75.78
Cash Collections	Small	31.24	190.10	76.41

Table 5.4 shows the performance results of the same configurations as before but with double the *datanodes* for a total of eight.

Table 5.4: Run-time (seconds) performance of MySQL server running on master node and Phoenix and Hive running on master node plus **8 datanodes**

Stored Procedure	Query Type	MySQL	Phoenix	Hive
All Operations	All	47.35	Timed Out	58.80
All Operations	Half	14.34	56.44	58.7
AllOperations	Small	20.59	50.66	58.23
Cash Collections	All	58.74	337.96	56.78
Cash Collections	Half	31.25	224.33	55.84
Cash Collections	Small	31.24	212.55	54.55

MySQL outperforms Hive and Phoenix for every query except the Cash Collections stored procedure equivalent for the "all" query type. Hive outperformed MySQL by approximately two seconds when using eight *datanodes*. The data set was relatively small, so it wasn't surprising that MySQL outperformed the others. The results for Phoenix were disappointing, but not surprising since it is a relatively new open source project. The Phoenix performance could be explained by its additional layer on top of HBase so therefore Phoenix needs more resources. The Phoenix All Operations and "All" query type timed out after ten minutes caused by not being able to obtain sufficient resources.

In all of the Hive runs for two, four, and eight nodes, the runtimes are consistent for each query type. For example, for the All Operations with two nodes, the times ranged from 85.50 seconds to 87.29 seconds and there is only a two second difference between the "half" query type and "all" query type. This remains true for all Hive runtimes, which is a good indicator that the runtime will remain relatively constant as the data sets increase in size. In contrast, Phoenix and MySQL runtimes increase with the amount of data. In the Phoenix Cash Collections stored procedure equivalent, the runtimes for "all" query type is almost double the half query type. In the MySQL All Operations method, the runtime for the "all" query type is about three times the runtime of the half query type. MySQL Cash Collections runtime for the "all" query type is about twice as much as the half query type. This is an indicator that as the data sets size increase, the runtimes will increase more dramatically for MySQL than Hive.

Table 5.5: Run-time (seconds) performance of MySQL server running on master node with double the data.

Stored Procedure	Query Type	Regular Time (sec)	Double Time (sec)
All Operations	All	47.35	113.63
All Operations	Half	13.34	27.06
AllOperations	Small	20.59	38.89
Cash Collections	All	58.74	118.53
Cash Collections	Half	31.25	60.95
Cash Collections	Small	31.24	57.18

To validate our hypothesis that the MySQL would increase more dramatically than Hive, we doubled our data set and ran the same runtime benchmarks on MySQL and Hive with the same configurations specified in the setup. We duplicated the data, assigning unique primary key values to the new data. For this experiment we only ran them for MySQL and Hive using the same configurations specified in the setup.

Phoenix was excluded because our previous results showed it is not a feasible solution. Table 5.5 shows the results for MySQL, where Double Time is the runtime on double the data set. Regular Time is the time to run on the original data set. Similarly, Table 5.6 shows the results for Hive. From these results, we can see that the times for MySQL doubled while the runtimes for Hive remained constant for HDFS running on four and eight *datanodes*. We can conclude that Hive takes approximately 55 to 60 seconds to get up and running.

Table 5.6: Run-time (seconds) performance of Hive running on master node and **2, 4, and 8 datanodes** with double the data.

# Nodes	Stored Procedure	Query Type	Regular Time(sec)	Double Time (sec)
2	All Operations	All	87.29	94.27
2	All Operations	Half	85.79	89.65
2	AllOperations	Small	85.50	89.66
2	Cash Collections	All	92.51	131.11
2	Cash Collections	Half	89.61	115.87
2	Cash Collections	Small	92.02	113.49
4	All Operations	All	58.54	67.51
4	All Operations	Half	62.10	62.76
4	AllOperations	Small	58.06	59.70
4	Cash Collections	All	85.09	86.16
4	Cash Collections	Half	75.78	76.07
4	Cash Collections	Small	76.41	79.80
8	All Operations	All	58.80	60.61
8	All Operations	Half	58.7	60.43
8	AllOperations	Small	58.23	61.39
8	Cash Collections	All	56.78	64.23
8	Cash Collections	Half	55.84	60.30
8	Cash Collections	Small	54.55	61.08
16	All Operations	All	-	59.77
16	All Operations	Half	-	58.97
16	AllOperations	Small	-	62.33
16	Cash Collections	All	-	60.15
16	Cash Collections	Half	-	56.94
16	Cash Collections	Small	-	55.33

In Table 5.7 we can compare the runtimes for MySQL and Hive using sixteen

datanodes with the doubled data sets. We can see that Hive outperforms MySQL for the All Operations stored procedure equivalent with "all" query type. Hive also outperforms MySQL with the Cash Collections stored procedure equivalent for all of the query types!

Table 5.7: Run-time (seconds) performance of MySQL running on the master node and Hive running on master node plus **16 datanodes** with double the data.

# Stored Procedure	Query Type	MySQL Time	Hive Time
All Operations	All	113.63	59.77
All Operations	Half	27.06	58.97
All Operations	Small	38.89	62.33
Cash Collections	All	118.53	60.15
Cash Collections	Half	60.95	56.94
Cash Collections	Small	57.18	55.33

In summary, neither solution is close to realtime, but MySQL outperforms Phoenix by a dramatic margin and Hive by a small margin. From Table 5.5, we can conclude that when the data sets double, Hive outperforms MySQL for the All Operations "all" query type with two, four, eight, and sixteen nodes. Hive also performed better with all of the Cash Collections query types with sixteen nodes and Cash Collections "all" and "half" query types for eight *datanodes*. Therefore, MySQL is the best candidate solution for this particular case study until there is a significant increase in data. Phoenix is definitely not a viable alternative, but when the data increases significantly, Hive proves to be a viable alternative.

Chapter 6

CONCLUSION

In this chapter, we summarize our findings, discuss the results and implications, and future directions for iVinci Health.

6.1 Summary

In Chapter 1 we introduced this research project. We provided a summary of what Marissa's project entailed and the results and timings of her project as well as briefly introduced the issues preventing the company from adopting her solution.

In Chapter 2 we give a background for BI concepts for storage and access. We also discuss the advantages and disadvantages for RDBMS. We summarized Big Data, which is likely to become a problem even for small to medium businesses. At the end we discussed Hadoop, MapReduce, Hive, HBase, Phoenix, and web interface concepts and relevant information that is necessary for this case study.

In Chapter 3 we introduced the case study for iVinci Health and the problems with Marissa's project and how we attempted to solve them in more detail. We discussed the data storage and access models, along with a description of how the data was retrieved using stored procedures. We also went over what the two iVinci Health stored procedures do and a general outline of the queries generated and executed.

Then we formally present the problem statement and discuss the goals and what we want to accomplish for this research project.

In Chapter 4 we defined the design and implementation for the RDBMS and DDBMS solutions. We discussed in detail the components for the MySQL, Phoenix, and Hive implementation of the solution. Finally, we discussed in detail what the web application solution entails and how each solution connected to JDBC.

In Chapter 5 we discussed a performance comparison for our experiment using RDBMS and DDMS. We discussed how the experiment was setup that included the version of the technologies used. Then we explained how we executed each solution and the steps taken. Finally, we presented the performance results for the experiment runs. We observed that neither Hive or Phoenix are able to outperform MySQL for this case study, but the Hive runtimes with the double data sets is evidence that this will change with a significant increase in data sets.

6.2 Results and Implications

From the performance comparison between MySQL, Phoenix, and Hive, we can conclude that:

1. Neither of the solutions were retrieved in realtime.
2. MySQL performs the best for this case study.
3. Phoenix performance was the worst and is definitely not a viable alternative to MySQL at the moment,
4. Hive performance remained constant for all cluster sizes and will most likely remain constant with larger data sets. The number of MapReduce jobs also remained constant for All Operations and Cash Collections procedures.

5. Hive will outperform MySQL as data sets increase dramatically in size.

Based on these conclusions we recommend to iVinci Health to keep their existing RDBMS solution until their data set increases.

6.3 Future Direction

It would be interesting to further investigate RDBMS and DDMS implementations to find a faster solution since the runtimes were not close to realtime for any of the solutions.

- **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. Hive– perform payment analysis benchmarks on various cluster configurations.
  2. Custom Distributed solution– It would be interesting to create our own implementations for HBase right join, union, etc instead of using Phoenix. We would like to implement them to be more efficient for our case study.
  3. Explore other Big Data technologies like Pig and MongoDB.
  4. Cluster– run the experiments on a dedicated cluster with full access and no resource limitations.

We would also like to generate larger data sets to at what point DDMS outperforms RDBMS.

REFERENCES

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

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

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

[4] L. George. *HBase: the definitive guide.* O'Reilly Media, Inc., 2011.

[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] HBase. `http://hbase.apache.org`.

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

[10] Marissa Hollingsworth. Hadoop and Hive as scalable alternatives to RDBMS: a case study. Master's thesis, Boise State University, 2012.

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

[12] JavaServer Pages technology. `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`.

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

[14] 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.

[15] Phoenix. `http://phoenix.apache.org`.

[16] M. Stepp, J. Miller, and V. Kirst. *Web programming: step by step.* Lulu, Inc., 2012.

[17] 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.

[18] 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.

[19] 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.

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

Appendix A

CLUSTER CONFIGURATION

A.1 Onyx Cluster Configuration

The benchmark experiments for the HBase and Hive solutions were executed on the Department of Computer Science, Onyx cluster at Boise State University. The cluster has one master node (node00) and 62 compute nodes (node01-node62), which are connected through a private Ethernet switch. Figure A.1 shows the layout of this cluster.

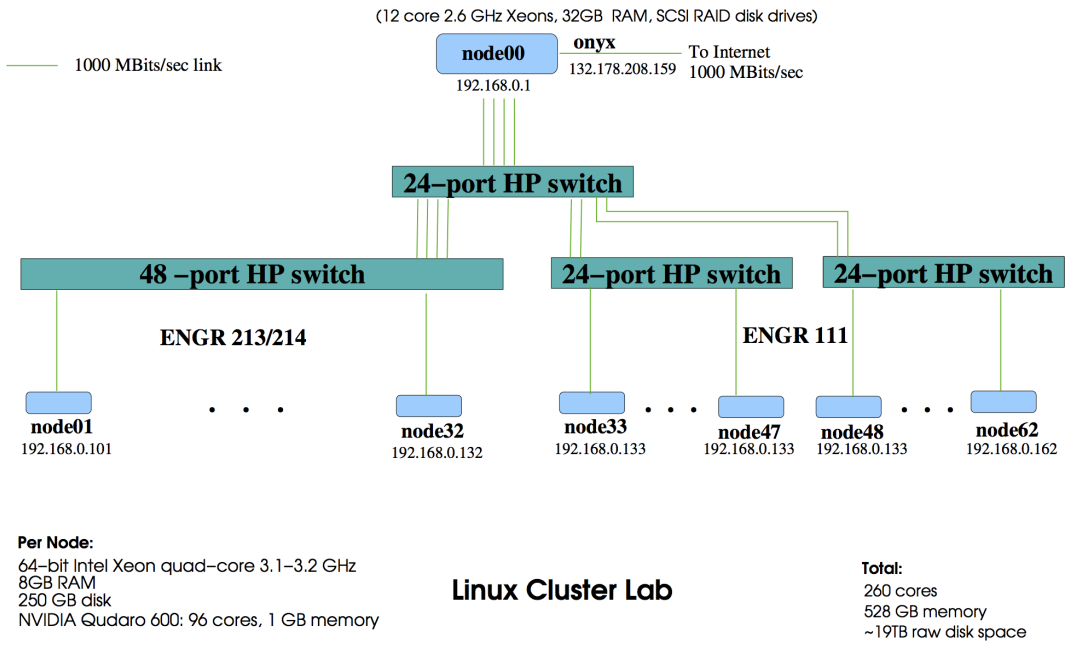

Figure A.1: Boise State University, Department of Computer Science, Onyx Cluster Lab

Master node (node00) is an Intel(R) Xeon(R) E5530 @ 2.6GHz processor with hyper-threading. Each node has 12 cores, 32GB RAM and SCSI RAID disk drives.

Each computer node (node01-node62) is an Intel(R) Xeon Quad-core 3.1-3.2GHz with 8GB RAM and 250GB disk. Each node has a NVIDIA Qudaro 600 graphics card with 96 cores and 1 GB memory.