A scalable database for a remote patient monitoring system

RUSLAN MUKHAMMADOV

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

A scalable database for a remote patient monitoring system

Ruslan Mukhammadov
ruslanm@kth.se

2013-07-18

Master of Science Thesis

Examiner and academic adviser
Professor Gerald Q. Maguire Jr.

School of Information and Communication Technology (ICT)
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Today one of the fast growing social services is the ability for doctors to monitor patients in their residences. The proposed highly scalable database system is designed to support a Remote Patient Monitoring system (RPMS). In an RPMS, a wide range of applications are enabled by collecting health related measurement results from a number of medical devices in the patient's home, parsing and formatting these results, and transmitting them from the patient's home to specific data stores. Subsequently, another set of applications will communicate with these data stores to provide clinicians with the ability to observe, examine, and analyze these health related measurements in (near) real-time. Because of the rapid expansion in the number of patients utilizing RPMS, it is becoming a challenge to store, manage, and process the very large number of health related measurements that are being collected. The primary reason for this problem is that most RPMSs are built on top of traditional relational databases, which are inefficient when dealing with this very large amount of data (often called "big data").

This thesis project analyzes scalable data management to support RPMSs, introduces a new set of open-source technologies that efficiently store and manage any amount of data which might be used in conjunction with such a scalable RPMS based upon HBase, implements these technologies, and as a proof of concept, compares the prototype data management system with the performance of a traditional relational database (specifically MySQL). This comparison considers both a single node and a multi node cluster. The comparison evaluates several critical parameters, including performance, scalability, and load balancing (in the case of multiple nodes). The amount of data used for testing input/output (read/write) and data statistics performance is 1, 10, 50, 100, and 250 GB.

The thesis presents several ways of dealing with large amounts of data and develops & evaluates a highly scalable database that could be used with a RPMS. Several software suites were used to compare both relational and non-relational systems and these results are used to evaluate the performance of the prototype of the proposed RPMS. The results of benchmarking show that MySQL is better than HBase in terms of read performance, while HBase is better in terms of write performance. Which of these types of databases should be used to implement a RPMS is a function of the expected ratio of reads and writes. Learning this ratio should be the subject of a future thesis project.

Keywords: *Big data, database performance, scalability, load balancing, Remote Patient Monitoring System*.

Sammanfattning

En av de snabbast växande sociala tjänsterna idag är möjligheten för läkare att övervaka patienter i sina bostäder. Det beskrivna, mycket skalbara databassystemet är utformat för att stödja ett sådant Remote Patient Monitoring-system (RPMS). I ett RPMS kan flertalet applikationer användas med hälsorelaterade mätresultat från medicintekniska produkter i patientens hem, för att analysera och formatera resultat, samt överföra dem från patientens hem till specifika datalager. Därefter kommer ytterligare en uppsättning program kommunicera med dessa datalager för att ge kliniker möjlighet att observera, undersöka och analysera dessa hälsorelaterade mått i (nära) realtid. På grund av den snabba expansionen av antalet patienter som använder RPMS, är det en utmaning att hantera och bearbeta den stora mängd hälsorelaterade mätningar som samlas in. Den främsta anledningen till detta problem är att de flesta RPMS är inbyggda i traditionella relationsdatabaser, som är ineffektiva när det handlar om väldigt stora mängder data (ofta kallat "big data").

Detta examensarbete analyserar skalbar datahantering för RPMS, och inför en ny uppsättning av teknologier baserade på öppen källkod som effektivt lagrar och hanterar godtyckligt stora datamängder. Dessa tekniker används i en prototypversion (proof of concept) av ett skalbart RPMS baserat på HBase. Implementationen av det designade systemet jämförs mot ett RPMS baserat på en traditionell relationsdatabas (i detta fall MySQL). Denna jämförelse ges för både en ensam nod och flera noder. Jämförelsen utvärderar flera kritiska parametrar, inklusive prestanda, skalbarhet, och lastbalansering (i fallet med flera noder). Datamängderna som används för att testa läsning/skrivning och statistisk prestanda är 1, 10, 50, 100 respektive 250 GB.

Avhandlingen presenterar flera sätt att hantera stora mängder data och utvecklar samt utvärderar en mycket skalbar databas, som är lämplig för användning i RPMS. Flera mjukvaror för att jämföra relationella och icke-relationella system används för att utvärdera prototypen av de föreslagna RPMS och dess resultat. Resultaten av dessa jämförelser visar att MySQL presterar bättre än HBase när det gäller läsprestanda, medan HBase har bättre prestanda vid skrivning. Vilken typ av databas som bör väljas vid en RMPS-implementation beror därför på den förväntade kvoten mellan läsningar och skrivningar. Detta förhållande är ett lämpligt ämne för ett framtida examensarbete.

Nyckelord: Big data, databas, prestanda, skalbarhet, lastbalansering, Remote Patient Monitoring System

Acknowledgements

I would like to express my gratitude and sincere thanks to my academic advisor and examiner **Prof. Gerald Q. "Chip" Maguire Jr.** for his valuable suggestions, extremely helpful feedbacks and indispensable recommendations. From the beginning to the end of my thesis project, he always supported me with his brilliant advices, kept me on a right track and helped me by sharing his magnificent experience.

My grateful thanks to **Tallat M. Shafaat** for helping me to choose suitable benchmark tools and his valuable support during the benchmarking of my thesis application.

Furthermore I would like to thank **Jim Dowling** for sharing computer resources and allowing me to benchmark my prototype.

Special thanks to my family, in particular my parents for their unconditional affection, moral support and lovely inspirations during the period of my study, and all through my life. I would not be the person who I am now without their support.

In addition, I would like to thank my friends and relatives. Their moral supports helped me to grow one pillar up and encouraged me to do the best whatever I do.

Table of contents

List of Figures

List of Tables

List of acronyms and abbreviations

ACID	Atomic Consistent Independent Durable
API	Application Programming Interface
BOINC	Berkeley Open Infrastructure for Network Computing
C2DM	Cloud to Device Messaging
CHF	Congestive Heart Failure
CPU	Central Processing Unit
GPS	Global Positioning System
CRUD	Create Retrieve Update Delete
CSV	Comma Separated Values
DDL	Data Definition Language
ECG	Electrocardiography
ETL	Extract Transform Load
GFS	Google File System
HDFS	Hadoop Distributed File System
HDVC	High Definition Video Conferencing
HiveQL	Apache Hive Query Language
HPC	High Performance Computing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corp.
ICU	Intensive Care Unit
I/O	Input/Output
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
IP	Internet Protocol
MATLAB	Matrix Laboratory
MPI	Message-Passing Interface
MS SQL	Microsoft Structured Query Language
MySQL	Structured Query Language
NoSQL	Not Only Structured Query Language
ODBC	Open Database Connectivity
OLTP	Online Transaction Processing
RAM	Random Access Memory
RCFile	Record Columnar File
RDBMS	Relational Database Management System
REST	Representational State Transfer
RMRS	Remote Medical Records System
RPM	Remote Patient Monitoring
RPMBS	Remote Patient Monitoring Backed System
RPMS	Remote Patient Monitoring System
SAN	Storage Area Network
SPOF	Single Point of Failure
SQL	Structured Query Language
UI	User Interface

URL	Uniform Resource Locator
VoIP	Voice over IP
WANDA	Weight and Activity with Blood Pressure Monitoring System
WHI PAM	West Health Institute Personal Activity Monitor
WHI SMS	West Health Institute Sign Monitoring System
XML	Extensible Markup Language
YCSB	Yahoo Cloud System Benchmark

1 Introduction

This chapter briefly introduces the area that is going to be investigated during this master's thesis project. It focuses on the main problem, how the problem is going to be solved, what the goals of this thesis project are, and how to achieve these goals. The final section of the chapter explains the overall structure of this thesis.

1.1 Overview

For many years the demand for health care of human beings has steadily increased. Additionally, more and more flexible and efficient ways of treatment are being developed. Today the use of information technology (IT) is wide spread in health care. One of the areas where it is being applied is remote patient monitoring. This application area is growing incredibly fast [1, 2]. According to a report from Berg Insight, around 2.8 million patients worldwide were using a home monitored service at the end of 2012 [3], and this number was expected to grow to 9.4 million by 2017 [4].

Remote Patient Monitoring is a technology that enables health care providers to monitor patients' health conditions while they are in their residences. This technology collects a patient's vital signs (e.g. blood oxygen saturation level, blood pressure), biometric data (e.g. level of glucose in blood, pulse oximetry), and other data (e.g. list of medications, diet compliance, disease symptoms). Today much of this data is collected by medical sensors and transferred electronically to a specific health care provider in real-time. This health care provider in turn provides an interface that allows clinicians and other health care personnel to monitor the health of their patients in (near) real-time. Some of the expected benefits of this technology are increased access to health care and a decrease in healthcare delivery costs.

Presently, a number of Remote Patient Monitoring systems (RPMS) offer a reliable solution which enables clinicians to monitor their patients [5, 6]. The majority of these systems use a traditional relational database management system (RDBMS) for data processing and storage. Although these systems manipulate the data through non-scalable and slow relational databases, they are currently able to effectively and rapidly handle this data. The primary reason that they are able to do so is because the number of patients that are being remotely monitored is still small. However, because of the rapid expansion of RPMSs, within a few years these systems will face a challenge managing the large amount of data that will be collected (and the large amount of data that was collected earlier). These collections of very large amounts of data are often called "big data". As a consequence of the expected increase in both the number of patients and the increase in the amount of data per patient that can (and will) be collected, there will be a need for scalability, high performance, load balancing, utilization of commodity hardware, etc. Unfortunately, it has already been shown that relational databases are not a good solution for handling big data [7, 8].

The definition of big data has been elucidated and described by a number of researchers and entrepreneurs. They all came to the same characterization. For example, Paul C. Zikopoulos, et al. [9] state that big data is structured and unstructured information that comes from everywhere, including various types of devices, social media sites, digital world, GPS signals, etc. IBM defines four characteristics or dimensions of big data [10]:

Volume terabytes and even petabytes of information;
Velocity high speed, non-delayed information exchange between source and destination;
Variety any type of structured and unstructured data; and
Veracity trust establishment among different kind of enterprises.

Efficient data processing and handling by fulfilling the first three dimensions is a real challenge for RDBMS, because when the amount of data increases, none of those three characteristics can be satisfied by relational database systems.

In an RPMS, big data could include any kind of health care related information, including health care measurements obtained from medical devices, video/audio communication between patients and clinicians, patients' health history, disease information, etc. Given this list of health related information we assume that the average amount of patient data stored *per patient* in RPM will range from a few megabytes to several gigabytes. This means that within a few years RPMS for a county (such as Stockholm – with ~2 million persons) will need to store Terabytes (10^{12}) to Petabytes (10^{15}) of information. Certainly new technology is needed for effective data processing and storing such a large amount of data [11].

Apache's Hadoop ecosystem provides a flexible and efficient solution for managing big data. It allows distributed processing of large data sets across clusters of commodity computers (nodes) using a simple programming model [12]. Clusters may contain a single server or thousands of machines, each providing storage and computing resources. The Hadoop architecture contains two core components: the Hadoop Distributed File System (HDFS) for storage and Map-Reduce for data processing. Both components are fundamental for a number of other components that deal with big data at various levels. This thesis project will design a new scalable remote patient monitoring system (RPMS) by implementing Hadoop with its components and then will implement and compare this prototype with an existing relational database oriented RPMS.

1.2 Problem Analysis and Definition

This section analyzes which technologies are the most appropriate choice for scalable data management and identifies the problems in the chosen area.

1.2.1 Problem Preview

Today's RPMSs can be divided into three parts where in the first part patients transmit their health measurements to the RPMS and by querying those measurements physicians access and monitor patients regularly. The second part contains only machines mining each obtained measurements and prioritize patients according to their medical records. Prioritization is strictly based on certain patterns which identifies the level of criticality on measurement results and gives highly prioritized patients to the third part. In the last part clinicians monitor only patients whose health is below than normal and therefore this part should function extremely quickly and provide real-time health measurement analysis. The main idea behind this thesis project is to investigate effective data management in the first part of today's large RPMSs and to design a new scalable system; therefore attention has to be paid to reliable data storage systems and efficient data processing.

Several techniques for building distributed systems have been proposed and implemented for big data handling. We will first examine the advantages and disadvantages of RPMS's current storage system (traditional RDBMS [13, 14]) and then examine replacing this using big data techniques. Following this we explain what data storage parameters have to be considered in order to build a scalable RPMS.

1.2.2 Traditional Relational Database Management System

A traditional RDBMS is a database management system (DBMS) based on a relational model. This has been the predominant choice for storing information in various databases. These databases are mainly used to manage the organization, security, access, and integrity of

data. The information is stored in a set of tables, each of which has a unique identifier called a primary key. The tables are then related to one another using a foreign key, which is simply a primary key in another table. Such relation oriented tables are effective in managing relational data.

Advantages of a RDBMS:

- Support for Atomic, Consistent, Independent, and Durable (ACID) transactions.
- Very fast for processing small data-sets – as these systems take advantage of hardware (latest Central Processing Units (CPUs), large memories, etc.) for processing;
- Implement Structured Query Language (SQL) – a special purpose query language that fits with any type of RDBMS;
- Comprehensive Online Transaction Processing (OLTP) support is really beneficial, especially for transaction-oriented applications;
- Privileges - full authorization and privilege control can be easily realized by the database administrator.

Disadvantages of a RDBMS:

- Cost – it is expensive to set up and maintain the database system.
- Implementing transactional, concurrency, consistency, and durability for large data-sets become very cumbersome as the size of the data set increases. Cluster based implementation is hard due to the nature of ACID.
- Unable to manage unstructured and/or semi-structured data as RDBMSs only work effectively with structured data. Lack of full support for unstructured and semi-structured data such as documents, videos, images, spatial data, etc.
- Scalability, clustering, and distributed realizations are hard as a RDBMS does not easily support distributed computing and clustering. Scalable data management is too slow.
- Fast text searching within fields is difficult.
- Some relational databases have limits on field lengths, which can lead to data loss if a data item is large.
- Making two databases, located in different areas, to "talk" to each other can be really expensive.

1.2.3 Distributed Computing Technologies

Many different distributed computing technologies may be used to replace a traditional relational database when the amount of data to process becomes enormous. Each technology fulfills certain requirements and provides a different level of efficiency when handling big data. This section introduces three well known and widely used distributed computing techniques that can be applied to big data.

1.2.3.1 Grid Computing: Message-Passing Interface (MPI)

Grid computing exploits a set of computer resources, potentially in different locations, to achieve a common goal [15]. Together with high performance computing (HPC), Grid computing provides large scale data processing by using message passing interface (MPI) APIs [16]. MPI is a message passing programming model which utilizes standard library functions on a wide variety of parallel computers. Broadly speaking, the main idea behind HPC is to distribute tasks across a cluster of machines, which access a shared file system. The shared filed system is frequently hosted by a storage area network (SAN).

Advantages of grid computing:

- Exploits parallel processing with different machines in the grid concurrently executing different parts of the task. This is a good choice for compute-intensive jobs.
- Virtual organizations can share their resources to form a large virtual computing system.
- The grid schedules grid jobs on computers with low utilization, thus achieves resource balancing to avoid unexpected peaks.
- Grid computing systems can provide reliability by using graceful recovery techniques to address an assortment of hardware failures. Processors, power supplies, and cooling systems are frequently duplicated so a failed subsystem can be replaced by another without turning the system off.

Disadvantages of grid computing:

- There are problems when nodes in the grid need to access large data volumes (hundreds of gigabytes). This occurs because network bandwidth becomes a bottleneck, and therefore compute nodes may become idle.
- MPI provides great control to developers; however, it requires them to explicitly handle the mechanics of data flow, using low-level C routines, sockets, and high-level algorithms for data analysis.
- When a failure happens in the grid, other machines may continue processing the other parts of the task without knowing about the failure. While MPI may allow control based upon failure detection, the code to do so is much harder to write.
- Grid computing computations may not be interoperable when different groups (with diverse components, policies, and mechanisms) want to share their resources.
- Shared infrastructure services should be provided to avoid repeated development, installation, and configuration – otherwise program development and operations will be slow.

1.2.3.2 Volunteer Computing

Volunteer computing is a type of distributed computing which enables ordinary Internet users to share their computer's storage and idle processing power as part of a high-performance parallel computing network [17]. This is a powerful distributed computing technique that can handle large amounts of data in an efficient manner by utilizing distributed resources. Volunteer computing is based on breaking the problem into chunks called work units, which are transmitted to idle computers around the world to be processed. When a client finish processing its assigned work unit, the results are sent back to a server and the client is assigned another work unit to process.

There are many platforms that achieve scalability through volunteer computing. Berkeley Open Infrastructure for Network Computing (BOINC) is an open-source software platform for computing using volunteered resources. It provides an opportunity for scientists to create and operate public-resource computing projects. A large number of diverse applications are used on top of BOINC to handle enormous processing power intensive research projects [18].

Advantages of volunteer computing:

- High-performance computing is possible by breaking the problem into independent pieces that can be processed in parallel using a set of machines.
- Resources can be shared among autonomous projects. This is facilitated because projects are never centrally authorized, thus each project operates its own servers. Volunteers can even participate in multiple projects.
- Because computer owners can registered with multiple projects, when one project stops or is closed for repair, another projects may inherit their computing power.

Disadvantages of volunteer computing:

- Different parts of the project are executed on untrusted machines connected to the Internet. These machines may have fluctuating connection throughputs and may not store data locality. The machines may also be removed from service or connectivity may be terminated at any time.
- Each work unit has to be sent across the network, hence the computational time should dwarf the transfer time, and otherwise the system will perform poorly. Because each volunteer donates CPU computing power the amount of computing power may not scale with the available aggregate bandwidth.
- Volunteer computing is not a good solution for private and proprietary applications as they are unable to rely on untrusted computing power shared by volunteers.

1.2.3.3 Apache Hadoop

Apache Hadoop is an open source framework to manage and handle large scalable data processing by writing and running various distributed applications [19, 20]. Distributed processing of a large amount of data is done in a Hadoop cluster (a set of parallel commodity machines networked together in one location). Millions of client computers can submit diverse tasks to this computational cloud and obtain results in a short time. Hadoop is also referred to as Key/Value Computing.

Advantages of Apache Hadoop:

- This solution is highly scalable as it distributes data across clusters of commodity computers and exploits parallel processing.
- A very large amount of data storage is available enabling scalability from a single node to hundreds of thousands of nodes in such a way that individual nodes can use local hard drives, processing power, CPU, and random access memory (RAM).
- Error handling is provided in the application layer, hence when a node fails, backup nodes can be added dynamically.
- When it is necessary to add more nodes to the cluster in order to make the system more powerful in terms of storage and performance, a few lines of refactoring code is sufficient to scale the machines. Unlike RDBMS, the Hadoop platform provides dependable performance growth proportional to the number of nodes available in the cluster.
- Hadoop distributes both data and computations, but computation is done only on local data preventing the network from being a bottleneck.
- Because all of the tasks are independent:
  - Partial failures can be easily handled by restarting entire nodes if they fail;
  - Propagation of failures and intolerant synchronous distributed systems can be avoided during data processing; and

- Speculative execution can be used to work around stragglers.
- Hadoop utilizes a simple programming model, hence an end-user developer only writes map-reduce tasks.
- When faults are detected by nodes, a quick automated recovery will be run immediately by the application layer.
- Each node automatically maintains multiple copies of data, thus in the event of failures, the copies of the data will be automatically redeployed and processed.

Disadvantages of Apache Hadoop:

- Hadoop is not yet mature and both Map/Reduce and HDFS are under active development.
- There is no central data, hence there is a restricted choice of programming models.
- Performing a "join" operation of multiple datasets is slow and tricky, as there are no indices. Often entire datasets must be copied in the process of performing a join.
- Cluster management is difficult, hence debugging distributed software and collecting logs of operations from the clusters is hard.
- The optimal configuration of (number of mappers, reducers, memory limits) nodes is not obvious.
- Managing job flow is not always trivial, as it is hard to manage flows when intermediate data *should* be kept.

1.2.4 Prerequisites for a Scalable RPMS

The basic requirements for a scalable RPMS are:

- The system has to support efficient handling of unstructured and/or semi-structured data, because most of the data in RPMS is not structured data (as it includes video/audio, individual health care measurements, etc.).
- Setting up and maintaining the data storage system should not be expensive.
- There is a limited need for data consistency; hence the ACID properties can be relaxed.
- The system has to maintain flat scalability, because as the number of patients grows the number of machines in the cluster has to be increased proportionally.
- Data storage should be sufficiently large to store and process all the incoming data.
- Data writing, searching, and retrieval for a single patient's data have to be very fast and independent of the number of patients in the system.
- Data will be written once and read several times. In fact, most of the data that is stored will never be modified or deleted; hence we can exploit this property of this data.

Generalizing all above requirements enables us to compare the above technologies in Table 1-1. Of all the systems shown in this table Apache Hadoop appears to offer the best solution for a scalable RPMS. All of these requirements need to be addressed during the development of our prototype.

Table 1-1: Comparison of Technologies with respect to their applicability for realizing an RPM

	RDBMS	Grid Computing	Volunteer Computing	Hadoop
Efficient support of semi-structured and unstructured data	No	Yes	Yes	Yes
Trustworthiness of system	Yes	No, because performing calculations on different organizational computers might not be entirely trustworthy	No, because computing power shared by volunteers might not be as secure as needed	Yes
Simple programming model	Yes	No	Yes	Yes
Data size	Gigabytes	Terabytes, mostly depends on supercomputers	Terabytes, mostly depends on server computers	Petabytes
High-performance processing of big data	Inefficient	Partially efficient because of the network bandwidth	Partially efficient because of the volunteer numbers in the project and network bandwidth	Very Efficient
Date updates	Read and write many times	Depends on the File System	Depends on the File System	Write once and read many times
Development and Configuration	Fast, requires less effort	Costly and difficult because written programs have to be run in supercomputers which have customized operating systems	Fast because of diverse applications	Very fast because of diverse applications
Scaling	Nonlinear	Linear	Linear	Linear
Structure	Static Schema	Dynamic Schema	Dynamic Schema	Dynamic Schema
Data Store Connectivity	Fast, because of data locality	Slow, because it is hard to provide constant high-speed network connection	Slow, because it is hard to provide constant high-speed network connection	Very Fast, because of data locality
Cost	Expensive because of non-linear behavior	Inexpensive because supercomputers do not need super hardware configuration	Inexpensive because main servers do not need expensive hardware	Inexpensive because ordinary computers may become nodes in clusters

1.2.5 Problem Definition

As said in the previous section, a scalable RPMS demands that we meet several specific requirements. During the designing process all of these requirements have to be considered. Additionally, some critical issues and missing functionality can be identified and should be investigated. The followings are some of the basic requirements and issues that were discovered and for which relevant solutions need to be found during this thesis project:

- All of the prerequisites mentioned in section 1.2.4 should be fulfilled;
- Currently Apache Hadoop provides a number of technologies for both HDFS and Map/Reduce tasks. A specific set of technologies should be chosen to fit the requirements of a scalable RPMS and to efficiently provide the needed function.
- For testing of two systems (RDBMS and Hadoop) real test data has to be collected or similar generated and the two alternatives have to be properly benchmarked.

1.3 Goals

The primary goal of this master's thesis project is to define an efficient way to build and operate a reliable, highly scalable, fault-tolerant, redundant, and highly available RPMS. After theoretically and practically proving the inefficiency of a current relational database oriented system, the project proposes a design for a new RPMS that should function smoothly with any amount of data. In order to achieve the project's main goals, the project is broken down into several tasks. These tasks are:

- Implementing Apache Hadoop along with all necessary and suitable components to build a scalable RPMS.
- Develop a scalable system that fulfills the requirements stated in section 1.2.4 and realize this system following professional coding standards and modern architectural design principles.
- Show that existing RDBMS oriented RPMSs cannot offer efficient data management for big data (in the case of an RPM). Test software suites will be used to benchmark both RDBMS and Hadoop systems and compare them in terms of performance, scalability, and load balancing.
- Clearly demonstrating the prototype system's reliability, scalability, fault-tolerance, redundancy, and high availability.

1.4 Methodology

This thesis project incorporates both qualitative and quantitative research methods. As the thesis project is based on a scientific study utilizing experimental and empirical approaches to demonstrate scientific validity, a quantitative research method was the primary technique. The main reason for choosing an experimental approach was the absence of a theoretical means to accomplish the goals stated in Section 1.3. Empirical and experimental approaches were utilized after clarifying the research question and identifying the evaluation metrics regarding performance analysis. In the end, a proof of concept prototype is to be implemented to answer the question posed. Due to the lack of access to actual health care data, generated data was used to test and evaluate the two different system architectures.

A qualitative research methodology was used to analyze earlier work done in this area. A design-based research approach is the most suitable technique to identify the limitations and issues of previous works and to design a new system. This approach provides new knowledge through the process of designing, implementing, and evaluating an artifact. In this thesis project, the artifact is an implementation of an Apache Hadoop platform. For the proper implementation of Hadoop, problems were specified in the first phase and then suitable literature was studied to characterize the implementation, deployment, and test cases in order to ensure that the relevant issues were examined. Eventually, a step-by-step procedure was proposed as a theoretical solution and a scalable system was designed and tested by following each step.

1.5 Structure of the thesis

The thesis project is organized as follows:

Chapter 1 introduced the idea of an RPMS. This chapter described current trends, limitations, and future expectations. It briefly analyzed the problem by comparing the advantages and disadvantages of traditional relational databases with three widely used distributed computing techniques. The problem was defined along with its context. The goals

of the thesis project were presented, along with the research methodologies that will be utilized.

Chapter 2 describes relevant earlier work, explains the overall architecture of RPM and analyzes its limitations and issues. By considering the advantages and disadvantages of prior work, Chapter 2 illustrates how suitable solutions should be designed to properly address the relevant parts of an overall solution for the stated problem.

Chapter 3 presents scalable and reliable data management techniques by giving brief descriptions. The chapter explains the different ways of implementing these technologies and reviews their efficiency in terms of development and gives some specific use cases.

Chapter 4 gives a detailed explanation of the architecture of the proposed scalable RPMS to be developed during this thesis project. This chapter describes the overall structure of the system, shows the interactions between the technologies that have been used and provides instructions of how to use them to achieve the project's goals. Finally, the chapter shows how the big data characteristics were addressed in the proposed reliable and fault-tolerant RPMS.

Chapter 5 benchmarks, analyzes, and compares two different RPMS architectures. The same amount of data that would be managed by an example RPMS is used for testing both RDBMS and Hadoop systems. A comparison of these two systems is illustrated using charts and graphs.

Chapter 6 concludes the thesis, suggests some potential future work, and describes the economic, environmental, and social aspects of this work.

Appendix A presents a sample data generated for the benchmarking phase of our newly designed prototype. Since, any kind of the medical information is considered to be security sensitive, we developed an application that generates "fake" medical records.

Appendix B includes all measurement results obtained during benchmarking of our prototype and illustrates the results in relevant graphs.

2 Background

A number of traditional RDBMS oriented systems are used to implement RPMSs. This chapter introduces several traditional RPMSs, explains the efficiency of data processing in these systems, and describes potential expectations (inefficiency of scalable data management) as the number of patients in the system increases. Specifically, data sorting, writing, and reading are considered as the main barriers when the amount of data stored in the system becomes very large, hence these operations are used to show the main drawbacks of traditional RPMSs.

2.1 Traditional RPMSs

As the new trend of using RPMS in modern patient care has proven valuable in a short period of time, the majority of RPMSs are being developed to provide simple, fast, flexible, and sophisticated services to both patients and clinicians. Three different RDBMS oriented RPMSs are presented in this section. Each system's architecture is concisely and clearly described in order to explain later how this system reacts when the amount of data in the system becomes enormous. Specific descriptions and explanations of the health measurement data utilized in each of these systems are outside the scope of this master's thesis.

2.1.1 The Technology Transfer Alliance's Carenet Services project: Remote Medical Records System

The Technology Transfer Alliance's Carenet Team has stated: "The objective of Carenet is to establish a research infrastructure where more cost-effective solutions to already existing demands can be demonstrated as well as completely new systems supporting the transformation of the health care work procedures to increase quality and cut costs in the overall process. The user scenarios addressed include teleconferencing sessions among medical experts, e-learning and remote patient monitoring." [21]

2.1.1.1 Overview

The Carenet Services' Remote Medical Records System (RMRS) enables patients to stream their health care data to a web server where clinicians can access these data and can monitor the patients in (near) real-time [21]. Currently, RMRS supports two kinds of medical sensors to measure the health conditions of patients. Additionally, Carenet Services provide reliable, robust, and fast high definition video conferencing (HDVC) to establish high quality video and audio communications session between patients and their doctors. In this system, three software applications were implemented to provide the services mentioned above. All of the information exchanged between patients and their clinicians are private and highly confidential.

2.1.1.2 System Architecture

The system includes three software applications to provide remote monitoring of patients. The first application is responsible for collecting health measurements from medical devices, performing an integrity check of the data, and securely transmitting the data to a web server over a network [22]. After receiving these data packets, the second application concurrently pushes the data to a web page to show these measurements to the relevant clinicians and stores the measurements in a relational database [23]. The third application provides high quality video/audio sessions between doctors and patients [24]. Figure 2-1 depicts the overall architecture of RMRS.

As shown in Figure 2-1, patients transmit measurement data to the server immediately after the measurement*. When the server receives the measurement data it automatically pushes the data to the relevant clinicians (if there is at least one open connection to a relevant clinician). Additionally, patients communicate with the relevant clinician(s) by exchanging video/audio packets. Communication session data is not stored on the server, this server only plays the role of a Session Initiation Protocol (SIP) proxy. All of the media within a session is directly transferred between the patient and the relevant clinician(s). Details of this multimedia media conferencing system are described in [24].

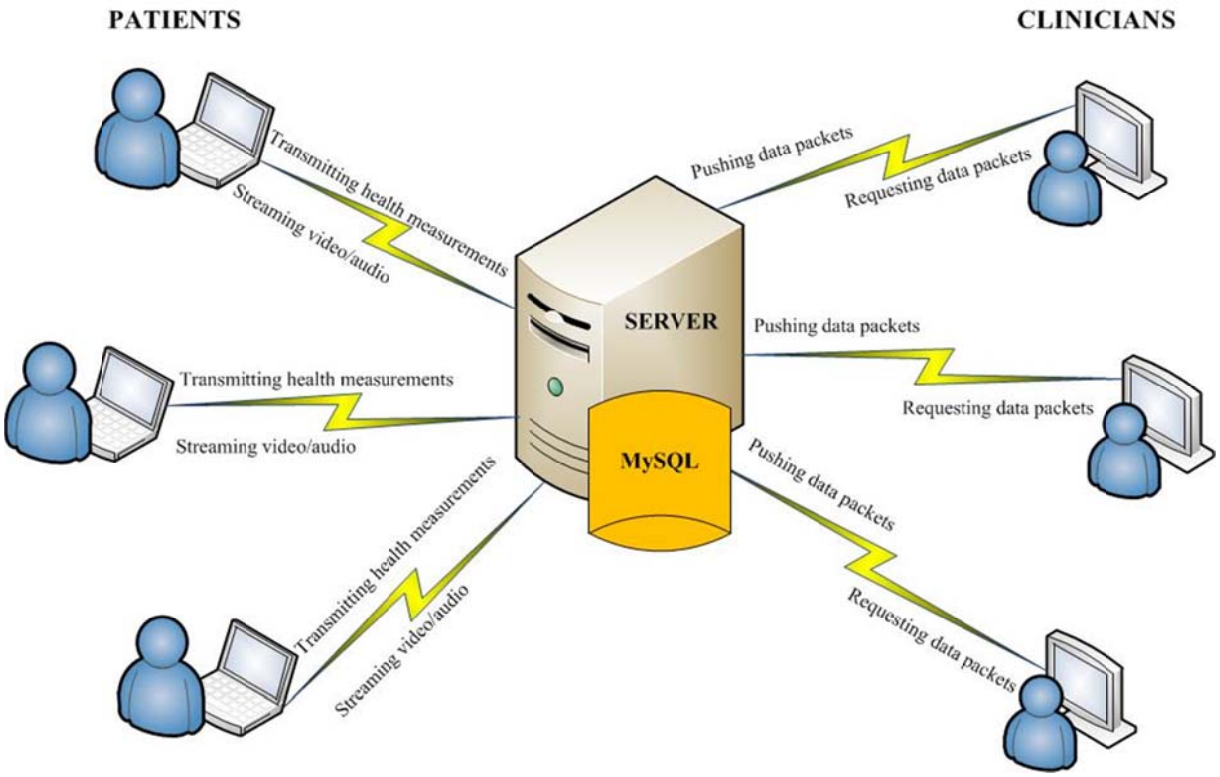


Figure 2-1: Overall Architecture of RMRS

Most users of the data stored in the RMRS are clinicians, rather than patients. The patients are concerned with the privacy and security of their personal health care data, therefore only the individual patient and their approved clinicians have access to the data uploaded by a patient. However, specific clinicians have a wide range of access to most of the patients' data in the system, as they need to access the history of diseases of patients, stored health care records, the treatments of past illnesses, etc. For these reasons certain types of requests will require more complex queries and supporting these queries will require more data to be processed and accessed at the database level. Details of what data a specific clinician can access are outside of the scope of this thesis, the interested reader may refer to Alexis Martínez Fernández's master's thesis to find out more about data access control of electronic health-care records [25].

* Details about the measurements and how they are made can be found in [22].

The second application handles SQL queries from the RMRS and it is considered the primary part of the system. The main responsibility of the application is to store all the health measurements, organize and present a patient's health care situation as one or more web pages, and retrieve the relevant data whenever a clinician requests it. This application utilizes a MySQL relational database to store the data obtained from patients. Nine database tables are currently used to store all of the relevant patient related information, including health measurements from different kinds of medical sensors, disease history of patients, earlier treatments of each patient, etc. Expensive SQL JOIN operations are used to process specific advanced SQL queries. The purpose of these queries are to retrieve patient specific information for a relevant clinician within a given date range with stated parameters; create, calculate, evaluate, and validate patient records; and to gather all relevant information (treatments, diagnoses, etc.) regarding specific diseases and/or patients. Details of these queries are given in [23].

2.1.2 Remote Patient Monitoring System

The Remote Patient Monitoring System (RPMS) proposed by Sherin Sebastian, et al. provides efficient tele-healthcare services by utilizing various hardware and software components [26]. The primary goal of the system is to enable cardiology healthcare services at a distance.

2.1.2.1 Overview

Various software and hardware tools were used to design RPMS. As stated above, the field of cardiology was the major focus for this system. Electrocardiography (ECG) was exploited for diagnosis. Signal processing techniques are used to analyze a constant stream of ECG signals (along with vital signs, various parameters, etc.). These signals were obtained as images from a display. Necessary information was extracted from these signals and examined with the help of MATLAB tools. All the processed information was then sent to a web server through an internet network. At the web server, all of the data was stored in a relational database and then pushed to the client for monitoring. As a result, clinicians were able to observe their patients' health conditions via a web page by using Android based smart phones and/or tablets.

2.1.2.2 System Architecture

In the initial phase of implementing the RPMS the system captured vital signs and other parameters from an Intensive Care Unit (ICU) which is positioned right in front of a webcam, the captured data was parsed, and meaningful information was extracted by analyzing the parsed data. The webcam continuously captures images at a rate of one image every four seconds from the screen of a bedside monitor. Each image contains information regarding to the patient's heart rate, ECG, blood oxygen saturation level (SpO2), and breathing rate. Digits in images represent the health conditions of patients and each digit is separately cropped out for analysis by utilizing MATLAB as a tool. The subtract function in MATLAB is used to compare a cropped digit image with a previously stored image in a MATLAB database in order to extract the image's numeric value. After image processing and analysis the data is uploaded to a web server by specifying the server's Uniform Resource Locator (URL) in the MATLAB commands.

At the web application server a MySQL relational database was used to store the health-care information extracted by MATLAB. The major tasks of the server include receiving a stream of data packets from multiple clients, storing the received data in a database, and composing this data into a web page which can subsequently be viewed using an Android

phone/tablet. The availability of near real-time health measurement data enables doctors to make appropriate diagnosis of their patients. Figure 2-2 illustrates the overall architecture of the RPMS.

The RPMS includes an additional server called a Cloud to Device Messaging (C2DM) server where Android phones register and obtain a unique identifier (ID). In critical situations, such as when a patient's current heart rate is outside the normal range, a batch file running in MATLAB generates an additional text file from the health information and transmits this text file along with the health data to the web server. When the server detects this text file it forwards this text file together with the corresponding ID of the clinician's Android phone to the C2DM server which sends a notification directly to the doctor's phone.

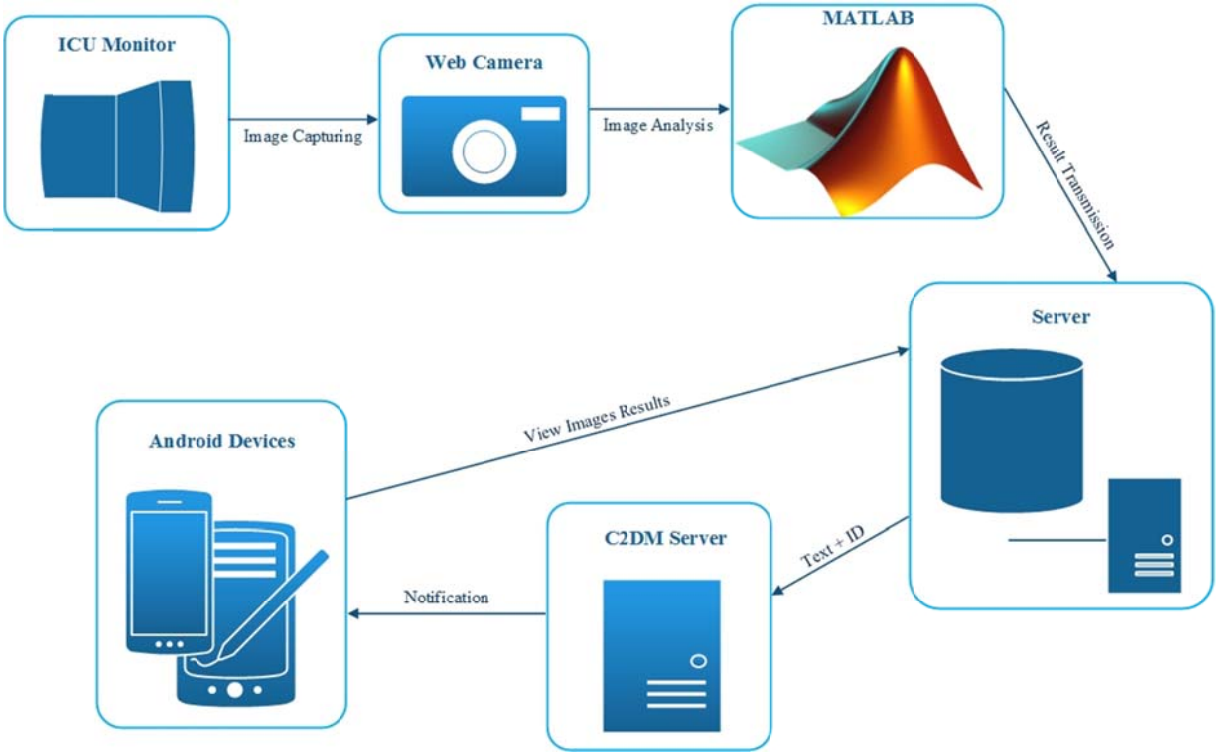


Figure 2-2: Overall Architecture of RPMS

2.1.3 Remote Patient Monitoring for Congestive Heart Failure

Remote Patient Monitoring for Congestive Heart Failure (CHF) is a system that provides a Weight and Activity with Blood Pressure Monitoring System (WANDA) architecture which leverages medical sensor technologies and wireless communications to monitor health related measurements of patients with CHF [27]. The WANDA system comprises of three major components which are sensors, a web server and a back-end database.

2.1.3.1 Overview

The core component of RPM for CHF is a WANDA system that was designed using a three-tier architecture. The first tier consists of medical sensors for monitoring patients' health conditions. All the health measurements are wirelessly transmitted to the second tier where a web server stores the measurements, checks their integrity, and push these measurements to a web application and/or a mobile application to enable the patient to be monitored. To maintain the availability of healthcare measurements, the second tier also forwards them to a third tier (a back-end database server). The main responsibility of the third tier is to perform database backups and recovery. Furthermore, the data in the third tier is used for various kinds of data analysis including linear regression, early adaptive alarms, missing data imputation, signal search, data security, etc. Figure 2-3 shows the overall architecture and the components interaction within this WANDA system.

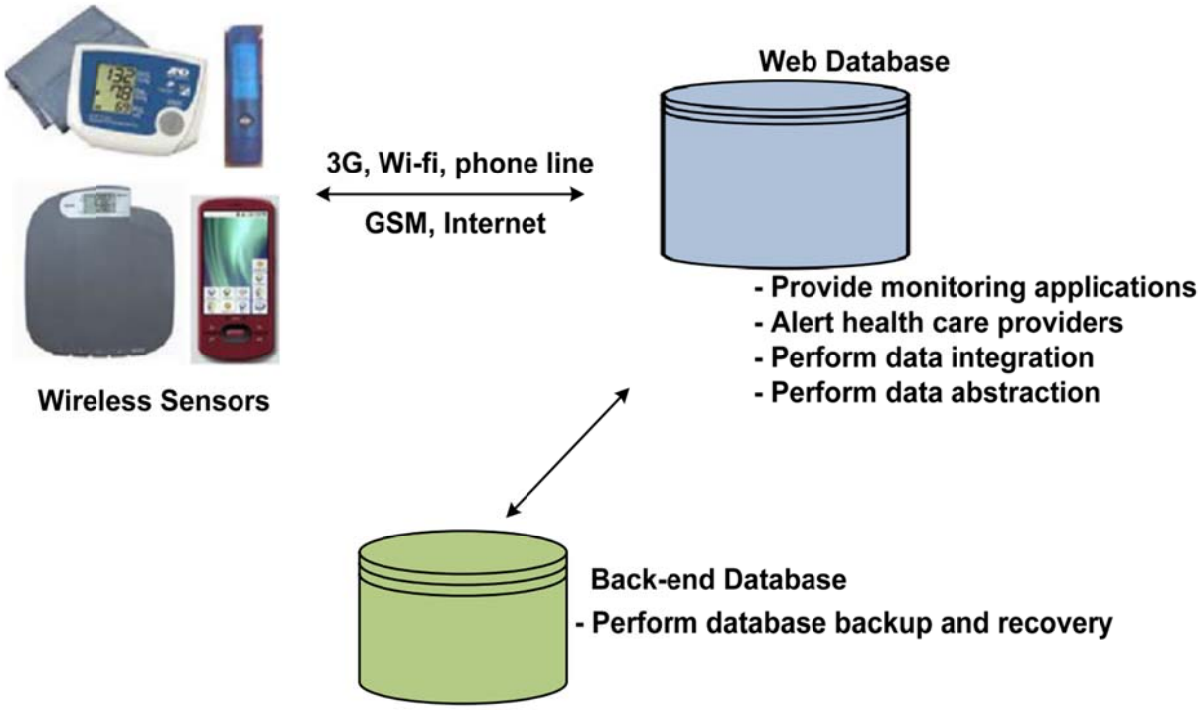


Figure 2-3: WANDA System architecture

2.1.3.2 System Architecture

The first tier in WANDA architecture is called the sensors tier and it is comprised of wireless sensors and mobile devices. This layer has two design versions/interfaces for two specific types of patients. The first version uses simple Bluetooth-based devices (such as weight scales, blood pressure monitors, personal activity monitors) together with a standard phone line connection for elderly persons who are not accustomed to smart phones and computers. In this version the connection between the first tier and the second tier is

established through a telephone system in real-time. Another version of the WANDA interface is much broader than the first one and it consists of smart phones which are able to collect and transfer health measurement* data to a web server.

Since the sensor tier is divided into two parts, the web server tier should provide appropriate processing for the data that are coming from two different types of sources and stores this data in a Microsoft SQL database. The reason for needing two different types of web server tiers is that the format of the data transmitted from the phone line version is much different than the data from the mobile version. Smart phones in the mobile version encapsulate the measurement data in a certain format and send it to the web server where based on the different data packet structures of each sensor these messages are parsed and the resulting data is stored in an SQL database. Unlike the mobile version, the phone line version utilizes Ideal Life, WHI PAM (Personal Activity Monitor), and WHI SMS systems which use incompatible data types and different databases. Data format abstraction and shared ID tables were used to efficiently analyze and store data transmitted using incompatible formats and coming from different databases. Another advantage of a shared ID table was to maintain data integrity of different types of medical records. In addition, when the received measurement results are out of normal range, the web server sends an alert message to a healthcare provider via an email or a text message. Details regarding data analysis, integrity maintenance, and specific SQL queries execution in the web server can be found in [27].

Even little items of medical data in an electronic medical record system can be vitally important; hence the data that has been collected should be actively guarded against data loss. The third tier in RPM for CHF, the back-end database tier, is responsible for performing data backups and recovery. Such a back-end database tier is incredibly helpful when it is necessary to restore a small number of files after they have been corrupted or accidentally deleted. The system uses the WHI SOPHI software to perform data backups and recovery. APIs† allow the SOPHI client application to communicate with a DBMS (Database Management System), to synchronize all new files, and to recover missing data.

2.2 Efficient data management in traditional RPMSs

The primary advantage of current RPMSs in terms of data management is the simplicity and flexibility of interaction with a RDBMS. SQL is a widely used programming language to query an RDBMS. Data authorization and authentication in these databases are managed in a very simple manner. Database setup and configuration does not require a specific operating system, as there are version of SQL capable databases available for most major operating systems and hardware platforms. Support of ACID transactions and Create, Retrieve, Update, and Delete (CRUD) operations are taken care efficiently. Additionally, third party database maintenance is available from a number of companies (some of which are quite large). Data processing in these databases is fast when the amount of data that is being processed is small.

Unfortunately, data management becomes a nightmare when the amount of data in a database becomes very large. Processing of even simple queries may take a long time. Furthermore, as the amount of data increases, the amount of commodity hardware needs to scale up faster than linearly, this leads to very expensive hardware configurations. Another big drawback of above mentioned traditional RPMSs is due to their attempt to manage both unstructured and semi-structured data through an RDBMS. This occurs because health measurements and much other patient related data are semi-structured and record-oriented,

* Details about the health measurements for both WANDA interfaces and how they are read from medical sensors can be found in [27].

† Details of the APIs developed and used in RPM for CHF system can be found in [28].

but an RDBMS is not a good candidate for such data. Additionally, all of these records are written only once, but read whenever clinicians or patients request them. Because all of the health measurements and history of patients' diseases & their treatments should be kept, there are no data updates or deletes required in the tables. Therefore after the initial write operation all subsequent operations are read only.

The requirements and drawbacks of current RPMSs lead us to define following set of basic issues that need to be addressed:

- The data store has to support advanced unstructured and semi-structured data processing;
- Data management has to be very efficient and independently of the amount of data in the system;
- The system should be fault-tolerant, highly available, scalable, and reliable;
- There is no need for data updates in the data store; and
- The system has to be able to scale out instead of scaling up (as scaling up would be too expensive).

All of the above assumptions become even more relevant as the number of patients in the system grows, potentially reaches millions of patients (or more). As a result, the amount of data stored in the system is expected to be terabytes to petabytes (or more), but responses to a patient's or clinician's simple query should be perceived as being nearly real-time (for the purposes of this thesis project we will set an upper bound of 2 seconds on the time to process a user's request – this time has been chosen as it represents the amount of time that an interactive user will accept as the delay for a response to a simple request).

3 Scalable Data Management: Apache Hadoop

This chapter introduces Apache Hadoop along with its major components which are suitable candidate components to build a new scalable RPMS. Not all of the components that will be introduced are needed for development of such an RPMS, as it is possible to design the system with a smaller set of tools that each is used to develop part of the system. The main reason why additional components are introduced is to understand and analyze the functional capability of each of these components and to provide more an efficient development path. Because, each component deals with a given task by using a different approach, there are different levels of development efficiency that can be achieved. Therefore, the components that are chosen should reduce the time needed for development, thus producing unique solutions for the tasks in their given area – while maintaining high interoperability with each other.

This chapter also explains one of the most important parts of a scalable data management system: Hadoop clusters. Assembling thousands of nodes into a single cluster and smoothly distributing tasks across these nodes requires great effort. Hence, appropriate cluster configuration and customization is vital for effective data processing. Lastly, Hadoop limitations and issues are examined. Designing a scalable system require deeper understanding of these limitations and their influences on the final system. Several methods are proposed to overcome these limitations. One proposed method was implemented in this design of a scalable remote patient monitoring backend system. Chapter 4 presents a detailed description of the Hadoop implementation that has been used to build a new scalable RPMS.

3.1 Apache Hadoop

In today's digital world, petabytes of information from millions of users are being processed every single day. Therefore, the primary concern for many large internet based enterprises is scalable and effective management of increasing amounts of data. Although RDBMSs have a long history of processing enterprise data, they are thought to be unsuitable for "big data". For this reason a number of new approaches and technologies have been proposed and implemented in order to redesign data management systems to meet the requirements of high availability, high scalability, and low latency; while maintaining application generality and exploiting weaker consistency requirements. One of the best-known and well-proven technologies that fulfill the above criteria is Apache's Hadoop ecosystem.

Hadoop is an open source platform that provides distributed storage as well as distributed computational capabilities. It is based on a distributed master-slave architecture which uses a simple communication model where one master process (a Name-Node) controls one or more processes called slaves (Data-Nodes) [29]. This architecture was initially inspired by Google's papers that described their novel distributed file system (the Google File System (GFS) [30]) and Map-Reduce [31] (a computational framework for parallel processing). The successful implementation of these two concepts resulted in a new technology that enables parallel computing and data partitioning of large datasets. The Hadoop Distributed File System (HDFS) and Map-Reduce were modeled after Google's GFS and Map-Reduce. These two are considered the core components of Hadoop for storage and computation, respectively. These two components scale with the addition of more and more nodes to a Hadoop cluster, and can reach an aggregate of petabytes of data on clusters with thousands of nodes. Yahoo![*] utilizes

[*] Yahoo's biggest clusters contain over 4 000 nodes each [32].

Hadoop to scale out very efficiently. Figure 3-1 illustrates the high-level Hadoop architecture with core functionality.

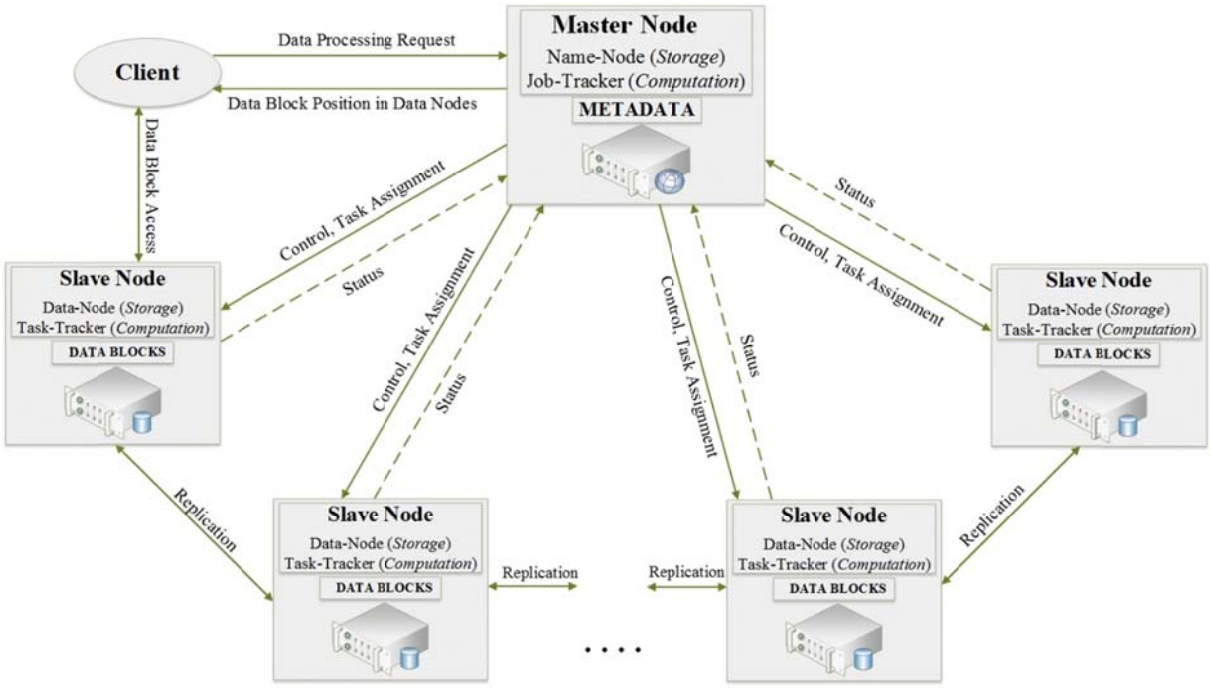


Figure 3-1: High-level Hadoop Architecture

Hadoop is written in the Java programming language and has been designed & is used by a global community of contributors. Although Hadoop itself is written in Java, any programming language can be used with Hadoop's Streaming APIs to run Map-Reduce jobs inside any specific system. There is wide support via a variety of libraries and extensions for other programming languages. A set of APIs* make all of the features of the Hadoop platform available to other programming languages.

In general, Hadoop is an inexpensive and very powerful distributed computing tool offering effective data management. It avoids costly transmissions of large datasets by using distributed storage and transferring code rather than data. Furthermore, data redundancy allows recovery in milliseconds when nodes in a cluster fail. Data partitioning, communication between nodes and node control are managed by the system *without* the intervention of application developers.

3.2 Major Components of Hadoop

This section describes Hadoop's basic components which handle RPM tasks in a very efficient manner. Some of them provide fast data processing, while others enable effective application development. The data format used in the system matters since certain components behave differently when processing various types of data. For these reasons, one component may be substituted for another based on specific criteria at the boundary of an efficient scalable RPMS. Furthermore, interaction between the components should be secure, fault-tolerant, highly-available and fast. Figure 3-2 shows the Hadoop Ecosystem with its major components. The figure clearly illustrates how the components interact with each other to build a scalable system.

* Details about the APIs for specific programming languages can be found in [33].

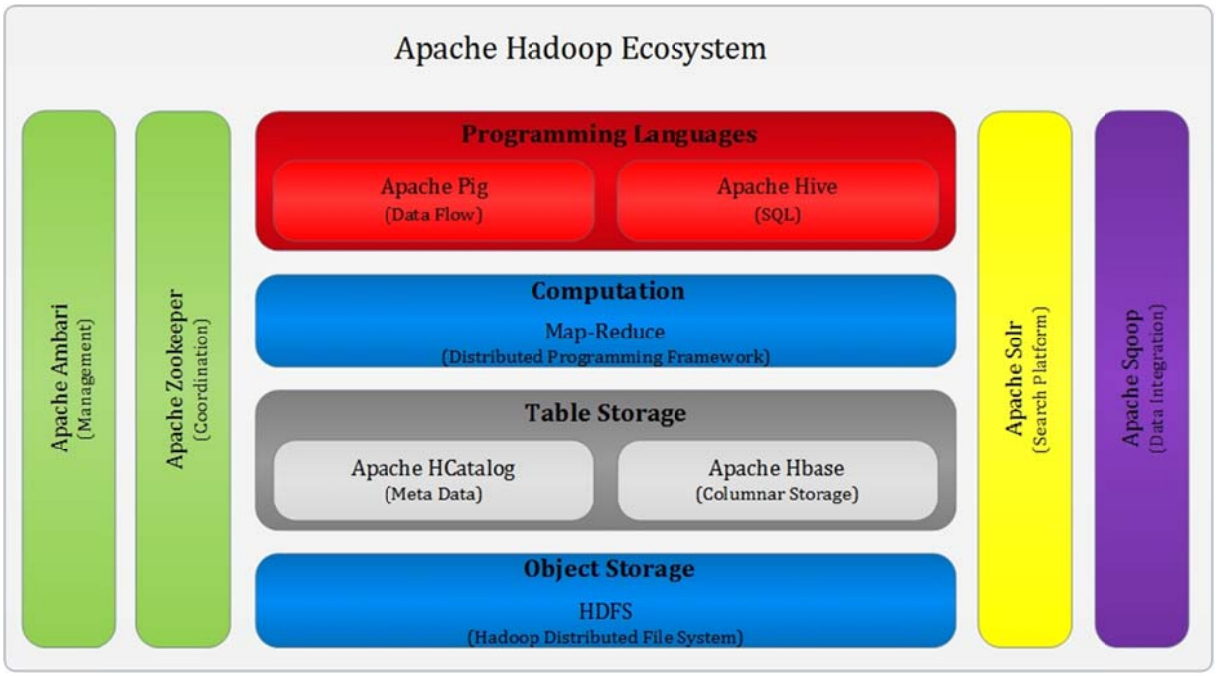


Figure 3-2: Hadoop's system architecture showing technologies relevant to this thesis

3.2.1 Hadoop Distributed File System

When a dataset exceeds the limits of the storage capacity of a single physical machine, it becomes urgent to distribute this data across a number of other machines. A file system that manages file storage across a network of machines is called distributed file system. Hadoop's core component for storage is the Hadoop Distributed File System (HDFS). HDFS is optimized for high throughput by leveraging large block sizes (by default 64 MB) and data locality improvements to reduce network input and output (I/O) [29]. Therefore, HDFS is one of the best options when it comes to reading and writing large files (gigabytes and larger). It is designed for the most efficient data processing pattern: write-once and read-many-times. For this reason, data modification in HDFS is impossible. Other well-known key traits of HDFS are reliability, scalability, and availability; and all of these attributes are achieved due to data replication and fault-tolerance. More specifically, HDFS mirrors the data to multiple (by default 3) storage nodes and automatically re-replicates the data blocks when nodes fail (both hardware and software failures were considered).

An HDFS cluster contains two types of nodes operating in a master-slave architecture [33]:

- A single Name-Node - a master node that stores metadata about the file system, and manages & controls slave nodes. A Name-Node provides a file system tree and stores metadata concerning all the files and directories in this file system tree. Through the metadata the Name-Node knows about the data nodes that store the data blocks of a given file.
- Multiple Data-Nodes – the slaves create, delete, and replicate data blocks upon instructions from the name-node and retrieve data blocks when the blocks are requested. Periodically, data-nodes report to the name-bode with a list of what data blocks they store.

The name-node in HDFS is considered the most important node and its failure may cause a disaster for a system, as having data-nodes without a name-node is analogous to a body without a head. Therefore, in order to maintain high reliability Hadoop provides two mechanisms for backing up the name-node data:

- Hadoop can be configured in such a way that a name-node writes its persistent state to multiple physical file systems, thus if a name-node fails, this data can be easily recovered.
- A separate physical machine called a secondary name-node can be used. This secondary name-node continuously merges image files and log files periodically from the name-node and keeps a copy of the metadata stored in the name-node. This secondary name-node can be substituted for the name-node when the name-node fails. Currently the use of a secondary name-node is deprecated and similar technologies called a checkpoint node or a backup node should be used instead [33].

3.2.2 Map-Reduce

Map-Reduce is a simple programming model for fast data processing. Map-Reduce parallelizes work over large amount of raw data on large clusters of commodity hardware in a fault-tolerant and reliable manner [29, 33]. The Map-Reduce framework is comprised of two basic functions: a map function (called mapper) and a reduce function (called reducer). A framework splits the input data-set into independent chunks, called blocks, and hands them over in the form of a key/value pair to a map function that processes them in a parallel manner. The map function produces zero or more key/value as output (which is simply an input to the reduce function). Along the way from the map output to the reduce input, the Map-Reduce framework performs shuffle and sort operations. These two operations are responsible for the following activities:

Partitioning	defining the reducer which must receive the map key/value pair; and
Rearrangement	sorting all its input keys for a corresponding reducer.

Shuffle and sort operations allow the reducer to receive combined and sorted values representing the same keys from the mapper. Each mapper output key belongs to a unique reducer as an input key; which means that the number of unique keys in the map output is the same as the number of reduce functions. Figure 3-3 illustrates a logical view of the map and reduce functions. The figure shows that all of the map output values emitted from all the mappers for the "key-2" key are collected in a "list (value-2)" list. Like the mapper, the reducer also produces zero or more key/value pairs as output. All these pairs can be written into flat files and stored in HDFS or insert/update rows in a NoSQL database. Basically, both the input and the output emitted from mappers and reducers are stored in a file system (or a database). The framework takes care of distributing the work, scheduling parts of the job on slave nodes, monitoring them, and recovering & re-launching failed tasks.

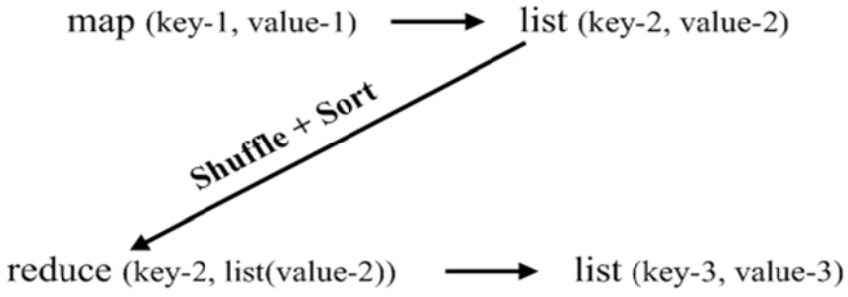


Figure 3-3: Logical view of map and reduce functions

Like HDFS, Map-Reduce also executes jobs in a master-slave manner: a single master Job-Tracker and one slave Task-Tracker per cluster-node. When Map-Reduce clients communicate with the Job-Tracker, the Job-Tracker starts scheduling jobs to be executed by Task-Trackers, monitoring these jobs, and re-executing jobs in case of failures. In other words, a master takes orders from clients and schedules map and reduce tasks on slaves to process these orders. The responsibility of a Task-Tracker, a daemon process, is to create child processes to perform a given map and reduce task.

3.2.3 Apache Hive

Apache Hive is a data warehousing technology built on top of Apache Hadoop. Initially Hive was developed by Facebook engineers to process large amounts of user and log data[34]. One of the great advantages of Hive is the creation of relational database-style abstraction that enables programmers to write in a dialect of SQL. Although SQL is not a good solution for big data problems, it is a great tool for data analysis. Hive's SQL like scripting, called HiveQL, is quite comfortable to use for developers who already have a good deal of knowledge regarding SQL, but who need to perform *ad hoc* queries, data summarization, and data analysis on big data.

Another huge benefit of Apache's Hive is that it provides a simple and quick way of writing Map-Reduce jobs. Because the Map-Reduce programming model requires the programmer to write code at a low-level, therefore developers tend to write custom programs which are very hard to maintain and reuse [34]. In contrast, HiveQL statements are able to execute jobs in both the map and reduce phases which are equivalent to Map-Reduce jobs. In terms of data analysis, Hive defines a table-like schema over a set of files stored in HDFS and extracts records from those files by executing HiveQL queries. The data in Hive is organized into three categories:

Tables	Hive tables are similar to relational database tables. Each table contains a corresponding HDFS directory where the data in this table is serialized and stored. In addition, Hive supports external data tables stored in HDFS or local directories.
Partitions	A partition is responsible for determining the distribution of data within sub-directories of the table directory. Each table can have one or more partitions.
Buckets	Buckets divide data in partitions. Division of data into buckets is based on the column's hash values in the tables.

In general, Apache Hive can provide the following features regarding scalable data management:

- Tools which allow easy data Extract, Transform, and Load (ETL);
- A mechanism to impose structure on a number of different data formats;
- Accessing the files stored in HDFS or other data storage systems such as HBase; and
- Executing queries via a Map-Reduce framework.

3.2.4 Apache Pig

Apache Pig is a platform for analyzing very large data sets and providing high-level data processing while retaining Hadoop's simple scalability and reliability [35]. Similar to Apache Hive, Apache Pig was created to simplify Map-Reduce jobs which are difficult to program. Map-Reduce allows developers to specify a map function followed by a reduce function and to follow this pattern programmers are required to write a number of Map-Reduce stages. In addition to writing the mappers and reducers, compiling the code, submitting each job and

waiting for results requires plenty of time which is unsuitable. Apache Pig avoids such complications by providing much richer data structures. It comprises of two parts:

- The Pig Latin language which is used to express data flows; and
- A compiler that compiles and runs Pig Latin scripts in a specific environment. Currently there are two types of environments: distributed execution on Hadoop clusters and local execution in a single Java Virtual Machine (JVM).

Pig Latin is a data flow language which allows developers to concurrently read and process data from one or more inputs and store results to one or more outputs. The data flows can be simple linear flows or complex workflows where multiple inputs can be joined to and split into multiple streams to be processed. In other words, data flows in Pig Latin can be considered operators which are applied to the input data to produce output. Taken as a whole, the Pig compiler translates the data flows into executable representations which are a series of Map-Reduce jobs, and then runs these representations.

A number of benefits make Apache Pig a widely used "big data" processing tool. Pig can process relational, nested, and unstructured data, can easily be extended to operate on data beyond files, databases, key/value stores, etc. Additionally, Pig manages all sorts of data - whether there is metadata or not. Another feature of Pig is that it is not tied only to parallel data processing, but rather it can be utilized in other types of data management. One of most important advantages of Pig for developers is that using it data processing requires only a short development cycle and it is simple to write Pig code. Pig allows the integration of developer code, hence it supports user defined aggregates, field transformation functions, conditionals, load/store functions, etc. These functions can be written in the Java programming language. Despite all of these advantages, there are few drawbacks regarding Apache Pig. Since Pig was designed for batch data processing (just like Map-Reduce), Pig is not a good solution to process a small amount of data in large datasets [32], because it is set up to scan an entire dataset or a large portion of corresponding datasets.

3.2.5 Apache HBase

Apache HBase is a low-latency, distributed, non-relational, column-oriented open-source database built on top of Apache HDFS. HBase also can be described as a persistent multi-dimensional sorted map, which is indexed by a row-key, column-key, and timestamp[36]. HBase is modeled after Google's Bigtable[37]. HBase is the best choice when real-time read/write random-access is needed to very large datasets. HBase can be thought of as a data store that hosts very large tables containing billions of rows and millions of columns atop clusters of commodity hardware. Unlike relational data stores, HBase provides incredibly fast access to large scale data while maintaining parallelization across a cluster of machines. HBase scales linearly by adding nodes to the clusters. Although it does not support SQL, it overcomes RDBMS several problems, including operating on a large number of hosts, working with sparsely populated tables on clusters, etc. The following features make HBase and even more widely used distributed data store:

- Linear and modular scalability;
- Well-suited base classes for backing Map-Reduce jobs with HBase tables;
- Stringently consistent reads/writes;
- Configurable and automatic table sharding;
- Simple and easy to use Java API for client access;
- Bloom Filters and Block caches for real-time queries; and
- XML, binary data encoding options, and Protobuf supports via REST-ful (Representational State Transfer) web services.

HBase can store any type of structured, semi-structured, or even unstructured data. HBase utilizes dynamic and flexible data modeling; hence there is no restriction on storing any kind of data. In Hadoop, both HDFS and HBase can store data in different ways. The main difference between them is that HDFS is a distributed file system used to store very large files; however it cannot provide fast individual record lookup in these files. On the other hand, HBase provides very fast record lookups and updates to large tables while storing all the tables in HDFS [32].

The storage model of HBase looks like a typical database, but with extra dimension. HBase contains a number of tables each of which consists of rows. Each row has a unique identifier called a row key and each row is formed from any number of columns. Usually rows are sorted in lexicographical order by their row key. Several columns can form a column family and each column in this family has multiple versions with a distinct value contained in a separate cell. Each value in a cell is either implicitly timestamped by the system or can be explicitly set by users. All columns in the column family are stored together in the same low-level storage file, called an HFile, in HDFS. Column families are defined by a table schema during the creation of tables. In general, with one expression an HBase storage model can be represented as follows:

SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>

The first *SortedMap* in the above expression is the table that contains a *List* of column families. The column families contain another *SortedMap*, which characterize columns with their corresponding values.

The HBase architecture can be seen in the expression of tables and their associated regions. As column families, regions are contiguous ranges of rows stored together. HBase takes advantage of regions to achieve scalability and load balancing. When regions become too large, the system dynamically splits them up. They can be spread across a cluster of nodes which distributes the load producing scalability. Each region can live on a different node and can contain several HDFS files and blocks. In addition, there are region servers in HBase which serve regions. Each region will be served only by one specific region server and region servers can operate on multiple regions at any time. Taken as a whole, region servers, also called slaves, serve data for reads and writes. Additionally, there is a master who is responsible for coordinating the slaves, assigning regions to slaves, and detecting failures of region servers. In the Hadoop ecosystem, Apache Zookeeper is one of the best tools which can be used as a primary helper of HBase master for managing slaves [32].

3.2.6 Apache HCatalog

HCatalog is a table and storage management service for Apache Hadoop ecosystem. It provides table and metadata abstraction layer which means the data in HDFS can be seen as a set of simple relational tables each of which resides in database. According to Alan Gates, the designer of HCatalog, “As an integration tool, HCatalog enables interoperability through the table abstraction. It presents a common table layer to each tool and gives each user the same data model”[38]. The primary goal behind HCatalog is to reference data without using specific filenames, formats, or storage paths. By default, it supports RCFile (Record Columnar File), CSV (Comma Separated Values), JSON (JavaScript Object Notation), and SequenceFile formats. HCatalog includes following features:

- Providing a data type mechanism and shared schema;
- Maintaining smooth interoperability among data processing tools including Map-Reduce, Hive, and Pig; and
- Creating an environment so that users no longer need to be concerned with how and where their data is stored.

HCatalog is built on top of a Hive meta-store and includes the Data Definition Language (DDL) of Hive. It provides a separate read/write interface for Map-Reduce and Pig tools. HCatalog utilizes Hive's command line interface to provide metadata exploration commands and data definition. HCatalog is an environment where Hadoop components can share their dataset's metadata information. For example, when a Pig user finishes their Pig Latin scripting and writes the data, Hive users can see these data as if it was Hive data. By using HCatalog developers can easily share the data written by different tools and do not need to care about the data types and formats used by this data.

The data model of HCatalog is similar to HBase's data model. Data is stored in tables and tables are placed in a database. Tables also can be divided into several partitions based on unique keys. Each key represents one partition and each partition contains all of the rows that belong to the same key. Partitions hold records and they are multi-dimensional, but not hierarchical. Each record is divided into columns each of which has a name and data type.

3.2.7 Apache Zookeeper

Apache Zookeeper is an open-source, distributed coordination service for distributed applications [39]. It provides a set of tools to develop distributed applications that implement high-level services for synchronization, configuration maintenance, and groups/naming. Usually, writing and maintaining distributed applications is hard because of partial failures. In a distributed architecture, when a message is sent across the network between two nodes and the receiver node goes down or network fails, the sender is unable to know whether the receiver got the message, this may lead to a partial failure. The only way to learn the status of a message is for the sender to connect to the receiver and ask it whether a given message has been received. The Zookeeper tools enable developers to build distributed applications that safely handle partial failures.

Although distributed applications *can* be managed by customized coordination services rather than using Zookeeper, providing such services in an appropriate way is notoriously difficult. Customized services are prone to race conditions and deadlock errors, thus, Zookeeper should be used to implement coordination services rather than building customized coordination services from the scratch. The operation of Zookeeper is straightforward; all servers that make up Zookeeper services should know each other. These services will be run as long as a majority of the servers are available. Clients connect to these servers to get served. If a server fails, then the client will connect to another server.

The main characteristics of Zookeeper are:

Simple — Zookeeper provides a shared hierarchical namespace to coordinate distributed processes. This namespace is similar to a standard file system and it is comprised of data registers, called znodes, which look like files and directories. Unlike a typical file system, Zookeeper is not designed for persistent storage; hence its data is stored only in memory; and thus, Zookeeper can achieve high throughput and low latency.

Expressive	The primitives/tools in Zookeeper are considered a rich set of building blocks which enable a developer to create large coordination data structures and protocols.
Highly available	Zookeeper operates on a cluster of machines and is designed to be highly available.
Reliable	Zookeeper avoids creating a single point of failure into the system and therefore maintains reliability.
Replicable	Zookeeper is intended to replicate itself over a set of hosts (called an ensemble).
Orderable	Zookeeper stamps each update with a number which orders all of its transactions. High-level abstractions can be implemented using this ordering.
Fast	Large clusters of machines enable a Zookeeper application to perform well, especially when read operations are more frequent than write operations (the ratio between read and write operations can be 10:1).
Facilitator	Zookeeper facilitates loosely coupled interactions. It supports processes that do not need to directly know each other, hence it may act as middleware for processes to discover and interact with one another. A process can leave a message in Zookeeper so that the intended receiving process can read this message at any time; even after the first process becomes idle or shuts down.

3.2.8 Apache Ambari

Apache Ambari is an open-source project providing simple and effective software tools for managing, monitoring, and provisioning Hadoop clusters [40]. Ambari makes Hadoop a single cohesive platform by simplifying the underlying operations and hiding Hadoop's complexity. As a Hadoop management tool, Ambari presents an easy-to-use web user interface (UI) backed by REST-ful web services. This web UI allows Ambari to provide a single control point for examining, managing, and modernizing Hadoop services. In addition, Apache Ambari provides effective security and recovery services through APIs. Ambari can be categorized into four service parts:

Provisioning	The web UI provided by Ambari presents a step-by-step installation of Hadoop services across a cluster of any number of machines. It helps configure Hadoop services for the cluster.
Management	Ambari provides single point of management to start, stop, and reconfigure the Hadoop services across the cluster.
Monitoring	Ambari provides several monitoring tools to monitor and examine a Hadoop cluster. It leverages Ganglia for metric collection and Nagios for system alerts. An Ambari dashboard displays the health and the status of a Hadoop cluster.
Integration	All above three service capabilities can be easily integrated with Ambari REST APIs[*].

The Ambari architecture consists of two parts and each part contains several components. The first part of the architecture is an Ambari Server, while the second part is the Ambari Agents. The Ambari Server is responsible for controlling a Hadoop cluster and this server processes the commands sent by an Ambari Agent. The server contains a master, an API, a relational database, and an Agent Interface. Each agent sends commands to the Ambari Server to check the heartbeat of the master. After receiving commands, the Agent Interface transfers the commands to a master and the master sends command back to the agent. The time interval

[*] Details of Ambari's APIs can found in [41].

of two back-and-forth command packets between the master and the agent determines the heartbeat of the master. The API provides access to monitoring and metrics for a Hadoop cluster. An agent communicates with the master to retrieve necessary information for access and may also send data regarding its operations. Depending on the request, the master may communicate with a relational database in order to retrieve or store data in it.

3.2.9 Apache Sqoop

Apache Sqoop is an open-source software tool that provides efficient data transfer between Hadoop and structured data stores, such as relational databases [42]. The primary goal behind Sqoop is to enable developers to import data from an RDBMS into HDFS or HBase, process the data with Map-Reduce or other higher-level tools (such as Hive and Pig), and export the results back into a RDBMS. Sqoop enables developers to effective communication between Hadoop and relational data sources. For data import, Sqoop only needs a data schema of the database, then the data will be automatically processed and exported back by Map-Reduce, Hive, or Pig which operate in parallel (with fault-tolerance) across a cluster of machines.

The input to import data from an RDBMS is a database table. Sqoop reads the table row-by-row (in parallel), produces a set of files as an output of this import, and stores the resulting files into HDFS. Later on Sqoop may integrate the output files into Hive/HBase or simply perform conversions, compression, partitioning and indexing on them. Depending on the data in the relational database tables, these output files may be delimited text files with comma separated fields, binary, or SeqFiles (Sequence Files) containing serialized data. The output is manipulated with Map-Reduce or Hive tools in a distributed manner and the files are exported back to the relational database. The export process includes reading delimited files from HDFS, parsing them into records, create new rows in the corresponding database table, and insert the new records into those rows. The export process is done in parallel.

3.2.10 Apache Solr

Apache Solr is an ultra-fast open-source standalone enterprise search platform based on Lucene [43]. It is a mature software package. Solr possesses a number of capabilities including advanced full-text search, near real-time indexing, faceted search, geospatial search, a vast variety of document handling, database integration, etc. Solr is also popular as a scalable, highly-reliable, and fault-tolerant search server which provides distributed indexing, automated failover and recovery, centralized data management and configuration, load-balancing, and query replication.

Data indexing and searching with a Solr Server is incredibly simple because of its REST-like API. Initially, indexing is performed via JSON (JavaScript Object Notation), XML (Extensible Markup Language), CSV (Comma-Separated Values), or binary data formats over HTTP (Hypertext Transfer Protocol). The indexed data can then be requested by querying via HTTP GET and received in the form of JSON, XML, CSV, or binary data format. To provide ultra-fast searching for specific data, Solr is optimized for high volume web traffic. Qualities such as comprehensive Hypertext Markup Language (HTML) administration interfaces, flexibility to configure XML files, linear scalability, and extensible plugin architecture make Solr platform even more powerful. Detailed features of Solr can be illustrated as follows:

Schema	Defines the fields of documents and the type of each field. Fields are responsible for certain functions. For instance, dynamic fields provides on-the-fly addition of new fields, copy fields enables to index a single field in multiple ways, or joins multiple fields into a single searchable field.
Query	HTTP interface with configurable and customizable response formats (such as JSON, XML, CSV, binary, etc.). Solr Query can perform a vast number of different search operations, perform different types of sorting, can combine different queries, and executes and provides ranges filters.
Core	Dynamically adds and removes document collections without restarting. Provides custom index processing chains, allows customizable request handler with distributed search support, controls documents with missing parameters, etc.
Caching	Provides fast processing of data searches by caching instances of Query Results, Filters, and Documents. Enables lock free and high concurrency cache implementations, cache warming and auto-warming in background, fast and small filter implementation, user level caching, etc.
SolrCloud	Provides a centralized Apache Zookeeper based configuration. Enables automated distributed indexing/sharding, near real-time indexing, transaction logs which guarantees no updates are lost, automated query failover, and no Single Point of Failure (SPOF).
Admin Interface	Advanced monitoring tool to observe cache utilization, updates, and queries. Provides full logging control, text analysis debugger, dashboard which presents the status of nodes in a cluster, output debugging, etc.

3.3 Hadoop Clusters

A Hadoop cluster is a set of machines/nodes each of which shares its memory and processing power. The nodes in a cluster are not required to be homogeneous, which means they may have different sized memories, different CPU architectures, and run different operating systems. However, it is advisable to use a cluster running the same operating systems and with similar hardware capabilities, as the cluster administration will be a lot easier when the machines have similar hardware and software configurations [32]. When it comes to setting up and efficiently running an easy manageable Hadoop clusters – everything matters; i.e., choice of the machines, including the specific hardware, operating system, disk configuration, and network design.

Hadoop nodes can be classified into masters and slave/worker classes. Master nodes run critical cluster services and therefore should be more robust and more resistant to hardware failures. Master node crashes may result in very expensive losses for companies. Slave nodes on the contrary, are expected to fail often. By default Hadoop replicates data on three slave machines, hence the data can be accessed from other machines when a given slave node collapses or crashes. For these reasons, to reduce the proliferation of hardware profiles many administrators choose a single hardware profile for all masters and a single profile for all slaves [44]. Another important thing for Hadoop is to determine the number of machines in a cluster. Usually, the cluster size is based on the amount of storage required.

Apache Hadoop is primarily run on Linux distributions as the underlying operating system. Today, a huge number of production clusters are running on top of Linux distributions (including RedHat Enterprise Linux, CentOS, Ubuntu, SuSE Enterprise Linux, and Debian). The main reason for Linux distributions to be chosen as the operating system is that they provide enhanced administration software tools, high level security, an open-source platform, and support a wide range of hardware. Hadoop operates as expected on Linux's common file

systems (including *ext3*, *ext4,* and *xfs)* [44]. Based on the architecture and criteria of scalable systems, one of these file systems will be chosen as a file system for a Hadoop deployment.

Because of its simplicity, the Hadoop platform is thriving in the real-world. Hadoop does not require any specialized hardware or network protocols to function efficiently. It runs equally effective in both flat layer 2 networks and routed layer 3 environments. Choosing an appropriate configuration of the network for Hadoop's core components plays a vitally important role in order to manage big data in an efficient way. In order to fulfill the scalability requirements, HDFS primarily focuses on three forms of traffic: Data-Node block reports and heartbeats to the Name-Node, block data transfer, and client metadata operations with the Name-Node. The Map-Reduce cluster membership and heartbeat infrastructure are similar to HDFS. The Task-Tracker continuously generates a heartbeat by sending a small bit of information to the Job-Trackers to learn if they are up and running.

3.4 Hadoop Limitations

No existing technologies are able to cover and fulfill all the aspects and requirements of real-world applications. In Hadoop, despite its advantages and strengths, observations show that it also has certain limitations and weaknesses. There are dozens of areas where Hadoop is the best choice, while in some other area it may not be well suited. The choice of a Hadoop framework as a core architecture depends on the requirements in a given area.

Currently there are two major weaknesses that have been identified for HDFS and Map-Reduce: availability and security [29]. Although both areas are under rapid development and enhancement, they are still unable to meet certain requirements. All of HDFS and Map-Reduce's master processes are a SPOF (Single Point of Failure). When a master processes crashes or dies, the control over tens or hundreds of slaves may be lost. Security in Hadoop is disabled by default and the only security feature in it is HDFS file and directory-level permissions and ownership. Hence, malicious users can steal another user's identity or impersonate them; they may even kill another user's Map-Reduce jobs. Another limitation of Hadoop is that HDFS is inefficient when handling small files. Because of its optimization for sustained throughput, HDFS is unable to provide effective random reading of small files. Furthermore, due to its batch-based and shared-nothing architecture, Map-Reduce is not good for real-time data access. Jobs that need changeable data sharing or global synchronization are not a good fit for Map-Reduce.

Hadoop has to be chosen based on specific criteria and the requirements of the system which is being developed. In our development of a scalable RPMS, Hadoop along with its components were chosen as a core framework. Some of Hadoop’s limitations have been partially solved, while the others did not affect the RPMS at all. High availability support for Name-Node and Job-Tracker is available in the latest 2.x versions of Hadoop, hence we will use the latest version to provide high availability. In order to provide the expected level of security, the Kerberos network authentication protocol can be run with Hadoop [29]. As providing high-level security in a scalable RPMS is outside the scope of this master's thesis, the implementation of Kerberos in the system is considered as future work. The limitation of HDFS is overcome by using Apache HBase which provides efficient random reads of small files. A scalable RPMS does not require mutable data sharing or global synchronization, so the limitation of Map-Reduce do not affect to the proposed RPMS.

4 Scalable RPMS Architecture and Design

This chapter explains and illustrates a proposed new scalable RPMS architecture and design. The implementation and design process are supported by a set of technologies to meet the requirements, which were stated in Chapters 1 and 2. The set of technologies were described in Chapter 3. For our system development, we have chosen the most suitable Hadoop components of those introduced in Chapter 3 and illustrated in Figure 3-2. The primary requirements for the proposed system are to provide linear scalability, fault-tolerance, high availability, and reliability. Most importantly, the Hadoop components need to be loosely-coupled and interact efficiently with each other, and fulfill all these requirements while providing parallelization.

4.1 Overview

The scalable RPMS architecture utilizes several technologies, each of which is responsible for handling some parts of the incoming data. The combination of these components allows developers to process all of the incoming data in milliseconds and provides relevant outputs when requested. For this thesis project such a combination of components should efficiently process gigabytes of incoming data from millions of patients and push the data to clinicians for monitoring. Additionally, clinicians may request earlier medical records regarding their patients. No matter how many patients transmit their vital signs nor how many clinicians are monitoring and requesting patients' data, the system needs to process each incoming request from a clinician in a relevantly small amount of time. The amount of time within which this response should be received will be discussed in Section 5.5. For the purposes of this discussion here we will assume an upper bound of 2 seconds.

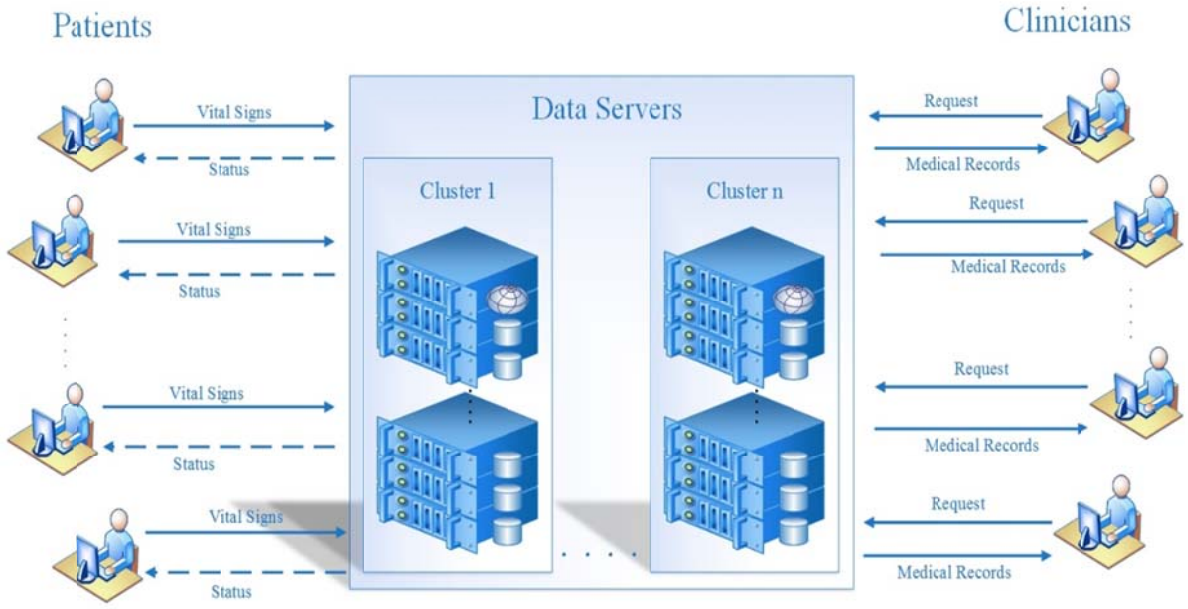


Figure 4-1: Overall Architecture of Scalable RPMS

Figure 4-1 illustrates the overall architecture of scalable RPMS. Data servers are considered the core of the system, because all of the data management is accomplished in these servers. The figure shows that as the number of patients and data increase in the system, commodity hardware will be added to scale out the system. Data servers may consist of several clusters and each cluster can include hundreds of machines. When a patient transmits

their vital signs, the data server processes this data and then sends a status message to the patient and pushes this patient's data to the relevant clinicians for real-time monitoring (if needed). Unlike patients, clinicians frequently communicate with the servers. Each clinician may request hundreds of patient records containing various kinds of medical information in one hour and must obtain relevant medical records within a short period of time.

4.2 Technology choice for Scalable RPMS

The Hadoop ecosystem provides tens of components to handle big data in an effective manner. Each component deals with a given problem in different ways and with different levels of efficiency. Therefore, the choice of a specific set of components to process scalable data depends on the system's architecture, data management strategies, and the structure of the data. In this thesis project the (limited) available time for development needs to be considered, since it is vital to design, implement, and evaluate a scalable system with high standards and high quality in the course of this thesis project. Figure 4-2 shows the set of Hadoop components which have been selected to develop a scalable RPMS.

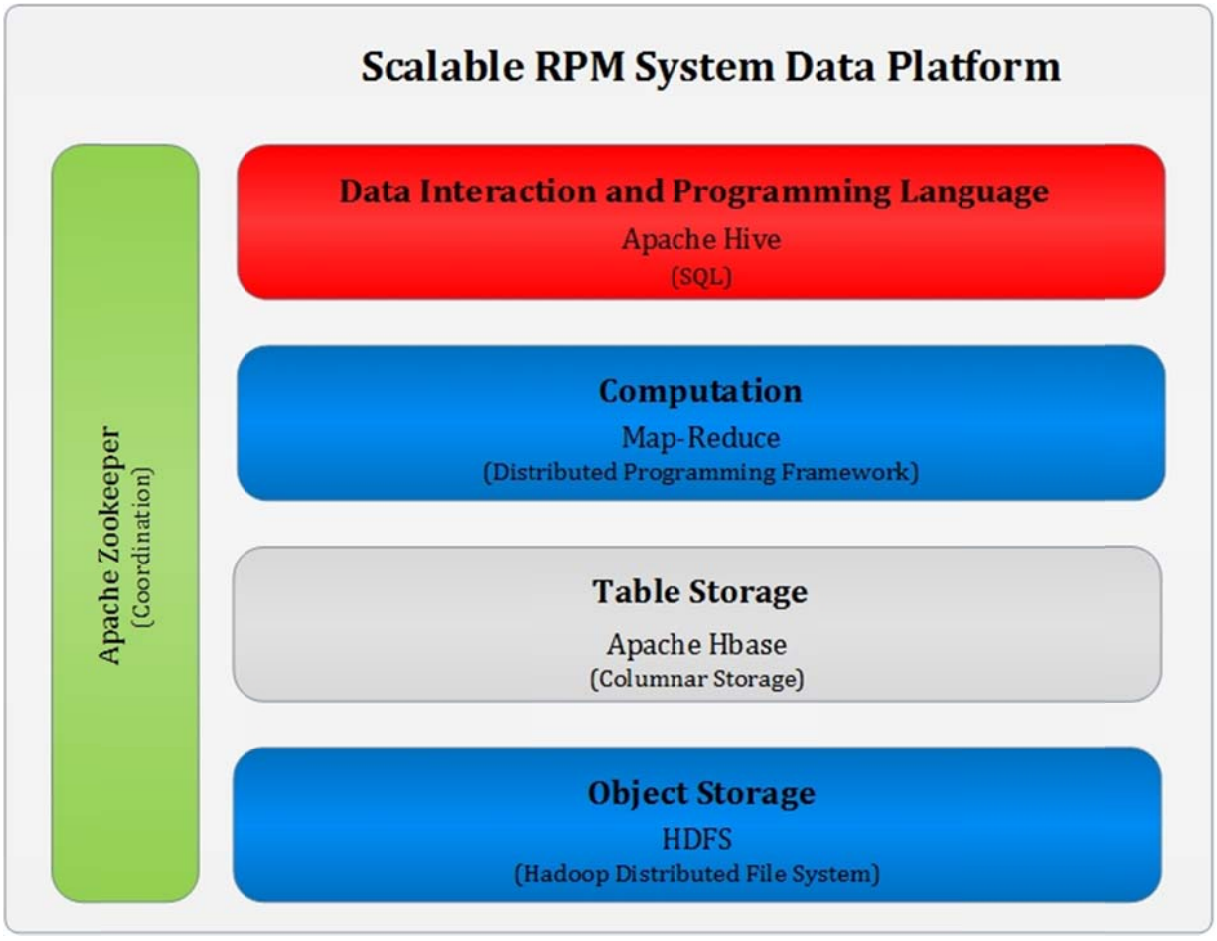


Figure 4-2: High-level RPMS data platform

It is possible to build a system by utilizing only Hadoop's core components (i.e., Map-Reduce and HDFS). Map-Reduce can perform data computations by distributing the tasks across a cluster of machines and store the outputs into HDFS. However, each system has its own data structures and requirements in terms of how the data should be stored and accessed. In RPMS, HDFS and Map-Reduce frameworks are not sufficient for efficient data

management and therefore several additional components are added to facilitate the implementation and operation of the prototype scalable RPMS.

As introduced in Chapter 3, Map-Reduce jobs are too complicated and time consuming for interactive use. Developers must write a lot of low-level code which might not be reusable. Additionally, code compilation, job submission, and result viewing takes a lot of time which hinders rapid development. To simplify Map-Reduce jobs, several complementary components were developed by other developers. Apache Hive and Pig are two such components and both are being successfully utilized by many enterprises. To handle Map-Reduce jobs efficiently, we have chosen Apache Hive over Pig. The primary reason for this choice is that Hive offers more features in terms of data access than Pig. Unlike Pig, Hive provides the notion of partitions. Partitions allow us to process a subset of the data based on specific criteria (such as date, alphabet, etc.). Additionally, Hive provides an optional Hive server, a web interface, and JDBC/ODBC driver for integration with SQL like databases[45]. Pig introduces a new language, Pig Latin, for data processing, while Hive utilizes SQL like scripting. As a result for a programmer who already knows SQL, it is easier to develop using Hive as this avoids the requirement to learn yet another language. This later aspect is important because learning a new language and utilizing it effectively requires great effort and additional time. As both (Hive and Pig) components provide facilities to achieve the same goals, we selected Apache Hive as a complementary tool for our Map-Reduce framework.

HDFS is not a good choice for some specific jobs. For instance, it is incapable of efficiently providing very fast random read/writes of a set of files. Additionally, HDFS is unsuitable for dealing with structured data and cannot provide effective individual record search over files. Fortunately, HBase, a component that is built on top of HDFS, solves these HDFS weaknesses. HBase provides very fast lookup and real-time random-access read/writes to large datasets allowing low latency access to a non-relational table structure. For these reasons, HBase was chosen to augment the functions of HDFS.

Zookeeper has been selected for use as a coordination tool in the RPMS. For patient monitoring, it is essential to handle partial failures. The sender must ensure that the receiver has successfully received the desired measurements or if the operation failed, then the sender should immediately initiate retransmission.

Apache Solr needs to be utilized as a search engine in the RPMS. Solr is able to provide ultra-fast search capabilities and a number of features that HBase cannot support. These features are important because RPMS has to deliver very fast record lookup to clinicians. Because of the shortness of period dedicated for thesis project, we could not implement Apache Solr in the RPMS and therefore marked it as a future work of this project.

Additionally, the implementation of Apache HCatalog, Ambari, and Sqoop are useful, but not essential to the realization of a prototype scalable RPMS. The future addition of these components will offer many enhancements and offer greater manageability of the RPMS. In addition, HCatalog and Ambari are not yet mature and they are considered members of the Apache Incubator project (a gateway for open-source projects that need to be further developed). For all of these reasons adding these components is seen as future work to *follow* this master's thesis project.

4.3 Tables Definition in the System

During the development of the application, we created two HBase tables to store all information regarding patient records and user data. The first table stores all user related information, while the second table stores all of the patient records (i.e., all of the health care measurements). Table 4-1 illustrates the first table. The unique row key is created by

appending the time of the record creation in nano-seconds and randomly generated 5 digit number. Users in the system may have various roles including patient, clinician, and administrator.

Table 4-1: User HBase Table

Row Key	Column Family	Timestamp	Columns
uid	data	T_1	first_name
		T_2	last_name
		T_3	username
		T_4	password
		T_5	email
		T_6	role
		T_7	cell_phone
		T_8	country
		T_9	county
		T_{10}	city
		T_{10}	street

Table 4-2 illustrates the second HBase table which stores all health care measurements obtained from patients. The row key of the table is constructed by appending a timestamp to the username. The timestamp for each table cell is provided implicitly, unless the developer enables it explicitly. New types of vital signs can be added on the fly as a new column to the table. Unlike RDBMS, HBase does not set a *null* value to the column when a corresponding value regarding to that column is missing. As a result, the number of columns in the column family may vary depending on the measurement results which need to be stored.

Table 4-2: Vital Sign HBase Table

Row Key	Column family	Timestamp	Columns
username + timestamp	vital_signs	T'_1	blood_glucose
		T'_2	body_pressure
		T'_3	body_temperature
		T'_4	end_tidal_co2
		T'_5	forced_expiratory_flow
		T'_6	forced_inspiratory_flow
		T'_7	gait_speed
		T'_8	pulse_rate
		T'_9	respiration_rate
		T'_{10}	spo2
		T'_{11}	tidal_volume
		T'_{12}	vital_capacity

4.4 The Distributed Storage Model

As said in earlier sections, Apache HBase is being utilized to store all of the patients' medical records. Based on the HDFS and HBase specifications, Figure 4-3 illustrates the distributed storage model of newly designed scalable RPMS. As the figure shows, HMaster runs on the Name-Node and controls all Region Servers on Data-Nodes. When new data arrives at the data servers, initially it is written to the HBase and then Solr subsequently performs the relevant indexing operation.

HBase heavily depends of Zookeeper to manage its slaves (Region Servers). Initially, clients communicate with a Zookeeper ensemble (a separate cluster of Zookeeper nodes) in order to retrieve the server name that hosts a particular table region containing the row key in question. After obtaining the server's address and the row key, the clients can directly communicate with the relevant region servers. There are number of ways for clients to communicate with HBase clusters. HBase native API, HBase Shell, REST, Avro, and Thrift are all great tools that enable effective communication between HBase clusters and various clients. For the development of a scalable RPMS, we have chosen two of them, HBase native API and REST. The main reason for their choice is that both tools provide the simplest and the fastest data structure models to build a scalable system (as comparing to other methods)[46]. In the first part of the application, REST is responsible for transmitting all measurements from patients and to retrieve relevant vital signs for clinicians. In the second part of the application, the HBase native API provides the application with the ability to create relevant tables with particular parameters. The table creation process is also bound to Apache Hive in order to take an advantage of its Map Reduce facility.

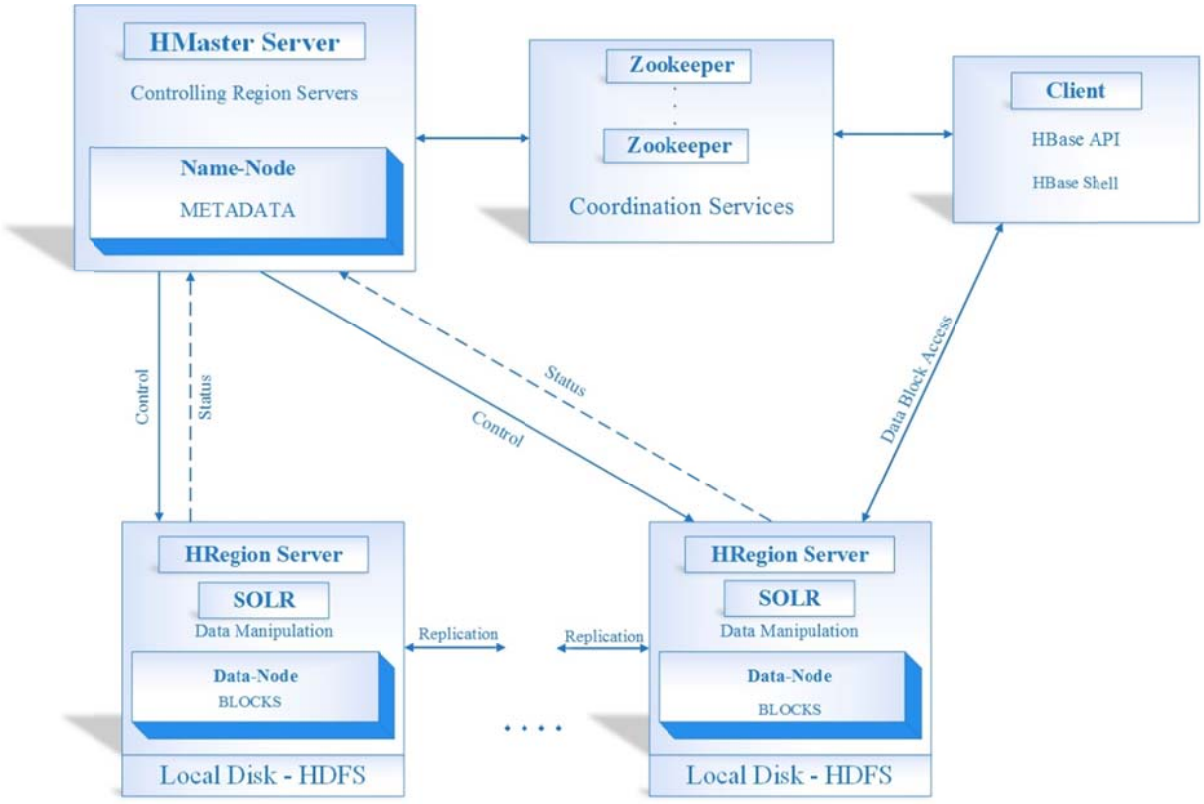


Figure 4-3: Distributed Storage Model of Scalable RPMS

4.5 Interaction of Hadoop Components

There are four key components in our scalable RPMS: HBase, Hive, Zookeeper, and Solr. Initially, the necessary tables are created in HBase using Hive and its native API. A HiveQL script provided by Hive, allows an application to create tables in Hive's own data warehouse as well as in HBase's store. Both data warehouse and storage are located on top of Hadoop's HDFS. Here is an example HiveQL script that creates a user table in Hive and in HBase with one column family and a row key:

```
CREATE TABLE user_hive (username STRING, password STRING,
firstname STRING, lastname STRING) STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH
SERDEPROPERTIES ('hbase.columns.mapping'=':key, info:val')
TBLPROPERTIES ('hbase.table.name'='user_hbase')
```

Subsequently when data is pushed to the *user_hive* table via the Map Reduce algorithm, the same data will be available when *user_hbase* table is requested. This interaction between HBase and Hive makes the system more robust. Additonally, the data analysis behavior of Hive becomes very helpful when analyzing large numbers of medical records and can enable an application to extract meaningful information in milliseconds. Here is a simple example script which retrieves all of the data in the *user_hbase* table:

```
SELECT * FROM user_hive
```

Note that even though the data is being read from the *user_hive* table, Hive actually retrieves all of the relevant data from the *user_hbase* table.

As explained in the previous section, Zookeeper ensemble tracks down all Region Servers of HBase cluster, lets clients to obtain information regarding to specific slave nodes, assigns tasks to slaves and reassigns failed tasks to another slaves. Each Region Server in HBase cluster continuously sends heartbeats to Zookeeper which allows Zookeeper to know exactly how many nodes are up and running. Advanced ultra-fast search capability of Solr helps clinicians to process complex requests in relevantly small time. Right after writing new incoming data to HBase, Solr indexing should be performed for each of them. Since we did not have enough time to implement Solr with its necessary features, it should be carried out in the RPMS as a follow up of this master's thesis project.

4.6 Application Source Code and Environment Preparation

Scalable RPM application includes several modern technologies mentioned in this section. The source code of the application can be found in here [47]. As the efficient data management is primary concern of this master's thesis, client side (reading health measurements from medical devices) development of scalable RPMS is outside the scope of this master's thesis. The application is divided into two parts:

- REST web services oriented HBase storage; and
- Apache Hive oriented HBase storage;

In the first part, HBase requires REST server to be ON. Clients communicate with HBase through REST server. On initial stage necessary tables will be created if they are not created yet and based on request and its type, specific operation (write or read) will be performed. REST receives all requests via light-weight JSON data interchange format, parses and processes them accordingly and sends back responses if necessary.

The second part of the application, is more statistics oriented, because Hive is able to perform complex operations through HiveQL queries. Likewise the first part, initially the tables are created and then specific queries are parsed. Two tables (‘user’ and ‘vital_sign’) are represented by two classes. Each class holds a logic belonging to the table it represents. This part also communicates with clients through JSON.

The application uses the core libraries of Hadoop version 1.0.4, HBase version 0.94.7 and Hive version 0.10.0. To run the application, it is necessary to install and configure Hadoop, HBase and Hive. Step-by-step Hadoop installation on a single-node cluster can be found in this guide [48] and step-by-step Hadoop installation on a multi-node cluster is described here [49]. HBase installation on a single node cluster can be found in here [50] and for multi-node cluster in here [51]. Full guidance regarding to Apache Hive installation and configuration can be found in here [52].

5 Benchmarking & Analysis

Benchmarking in this master's thesis project is divided into two parts: proper test data generation and system's testing with this generated data. Performance, scalability, reliability, fault-tolerance, and load balancing are key parameters that are considered in all rounds of this benchmarking. In a subsequent phase of the evaluation, these benchmarking results are analyzed and distinctions between the different alternatives are illustrated with charts and graphs.

5.1 Overview

Since the main goal of this thesis project is to design a scalable system that provides near real-time medical record data management and analysis, the benchmarks should be run in a way that obtained results from them have to clarify the relationship between the number of machines in the cluster and the amount of health data. Besides, the performance comparison between RDBMS and the scalable RPMS also should be considered during benchmarking. As Apache HBase was chosen as the storage model and tool to realize a scalable RPMS, we benchmarked and compared HBase against MySQL (as a representative RDBMS).

To achieve our goals mentioned above, we designed several scenarios and utilized them for testing. These scenarios include data input, data output, and computing statistics over the collected data. In all of these scenarios, different amounts of data are processed and analyzed with different parameters. In the first phase, MySQL data processing is benchmarked and the results are compared against HBase residing on a cluster consisting of a single node. Subsequently, the number of nodes in the RPMS is increased by one and the performance is once again calculated based upon the processing time required to process the same amount of data. The benchmark results are combined together and relevant charts are constructed in order to clearly show the difference between MySQL and HBase solutions. Most importantly these charts illustrate how efficiently the data is managed by two systems, how the new system behaved when the number of machines is increased, and how much new health data (corresponding to the number of newly added patients) requires adding an additional machine to the system.

In general, both systems store two kinds of data: user data and health care measurements. Once each user is authenticated by the system, he/she will continuously access the health care measurements table for different purposes. For instance, patients will write their latest vital signs; while clinicians may request large numbers of patient records, perform data analysis on the existing medical records, search for specific patient records collected during a specific range of dates, etc. In general, the number of accesses to patient records in the database is much greater than the number of appends of new patient data. We assume that this ratio is nine reads per write of a patient data record. For this reason the health care measurement table (with data about the vital signs of patients) is the primary focus of our system's benchmarking. Before starting benchmark, MySQL database (vital sign table in this case) is horizontally partitioned (in other words "sharded") [53, 54], since comparing single MySQL instance against several instances is not fair. Consequently, a sharded MySQL is benchmarked running on a single machine (it is highly efficient to split a large partitioned table over multiple logical database servers [55]). The proposed scalable RPMS is benchmarked with 6 machines with identical parameters (1 master, 5 slaves, and 1 computer used for data generation and transmission). A testbed was setup with one or more machines each with the configuration described in Table 5-1.

Table 5-1: Configuration of Each Machine Used for Benchmarking

CPU	Six-core AMD Opteron™ Processor 2435
CPU MHz	2592.994 MHz
Cache Size	512 KB
RAM	32 GB
Hard Drive	1 TB
Operating System	Ubuntu 11.04

5.2 Generating Test Data

Unless patients have given consent to someone to access their medical history there should be no access by anyone to their individual records, therefore all medical information stored in electronic databases is considered as highly sensitive personally identifiable data. Each system used for storing patient data has its own policy to securely store this sensitive information. For this reason and the lack of access to actual medical data, it is impossible to test the two systems (Hadoop and RDBMS) using real medical data, therefore we needed to generate electronic medical records similar to real world medical data. According to Health Level 7 Clinical Document Architecture (HL7 CDA) [56, p. 7], electronic medical records can be categorized into five types. Each of these types of records will need to be considered when generating test data. These five types are:

1. Basic information about each patient, typically: name, birthdate, gender, marital status, and contact information. The contact information might include mobile/home phone, fax, and/or e-mail address;
2. Previous disease history may include disease name, doctor, hospital treatment, treatment period, treatment procedure and results, history of infectious diseases, history of trauma surgery, history of significant drug use, history of allergies, and history of vaccination. The history of vaccination includes vaccine name, inoculation period, inoculation hospital, name of health care personnel who administered the vaccine, and vaccination results;
3. Physical examination may include heart rate/pulse, temperature, blood pressure, oxygen saturation (SpO2), respiration, posture, weight, etc.
4. Specialist examination may include pediatric, adult medicine, and other examination records; and
5. Medical information creation includes hospital name, creation date, and date of visit.

The prototype RPMS covers all five types of data. As the main goal of this master's thesis is to compare the efficiency of scalable data management systems with RDBMS, generating and testing one type of above electronic medical records was sufficient. Physical examinations (pulse rate, SpO2, etc.) were chosen for the benchmarking process as these types of data are expect to be generated much more frequently (perhaps as many as several times per day every day) than the other types of data. Based on this, we developed a program which generates "fake" physical examination medical records (a relevant range for each one of a number of vital signs is given and the software automatically generates a randomly value within that range as a hypothetic value representing a possible value for this record). This generated data is placed into a set of files. During testing we push data from these previously generated files into the proposed scalable RPMS. We calculate the throughput and latency based on the operation we performed. We will later compare the various alternatives by analyzing these throughput and latency results. The total volume of all of the generated data files was 254 Gigabytes (more than 250 million rows). This corresponds to 30 days (1 month) of data from ~150,000 patients which means each patient sent his/her measurements ~56 times in one day.

An assumption is that this amount of data should be sufficient to compare benchmarks of the alternatives in the various scenarios. Specifically, we generate data consisting of a *unique identifier* of a patient, *measurement results,* and a *timestamp*. Appendix A illustrates a sample from this generated data. The source code of the application used for generating necessary amount of data can be found in [57].

5.3 Benchmarking of the alternative Software Suites

There are number of tools to benchmark both MySQL and HBase. We have chosen the most common open-source tool, Yahoo Cloud System! Benchmark (YCSB) [58, 59], as it is easy to configure in order to generate benchmarks. Instead of using several benchmark tools we decided to utilize only one tool because YCSB is purely written in Java which makes it relatively portable, it appears to offer support for horizontal and vertical scaling in different ways, it is very easy to implement, it covers several workloads, and most importantly developers are not required to write many lines of codes to customize YCSB to benchmark their systems.

5.4 Yahoo Cloud System! Benchmark

In the scalable RPMS benchmarking process, data reads, writes, and statistical queries are used to extract data from or place data into the HBase while in traditional RPMSs MySQL was the primary data store (see Section 2.1). The YCSB suite has been utilized to benchmark both a RDBMS system and our prototype of the proposed system. YCSB is an open-source benchmarking framework which is intended to benchmark different types of data stores. The framework consists of a workload generating client and a package of standard workloads that cover the main parts of a performance measurement, such as read-heavy workloads, write-heavy workloads, scan workloads, etc. One of the great features of YCSB is its extensibility; specifically the workload generator allows developers to easily define new workload types, redefine the storage system that YCSB interacts with, and to adapt the client to benchmark new systems. YCSB's facilities allowed us to re-implement its basic interface in order to interact with HBase and MySQL. There was no need to define new workloads, because the existing workloads in YCSB were sufficient for us to benchmark our systems. Specifically, four types of workloads were utilized in the benchmarking process: workloadA which contains from a mix of 50/50 reads and writes, workloadC which is a read only workload, workloadD which is read latest workload, and workloadE which is range scan workload. Those workloads also can be interpreted in the medical records setting as follows:

- Workload A – at the same time patients are transmitting their health measurements clinicians are accessing this data;
- Workload C – only clinicians are reading data from the system;
- Workload D – clinicians are accessing the latest health measurements of specific patients and some number of patients are transmitting data packets; and
- Workload E – clinicians are retrieving statistics concerning a specific number of patients concerning a specific date range and at the same time some numbers of patients are transmitting their health measurements data.

YCSB gave us the same output parameters with different values for each workload. Specifically, by running the workloads that we need we obtained values corresponding following set of parameters:

- Overall runtime – elapsed time for the selected operation to be completed;
- Overall throughput – average speed of the system related to the number of operations per second;
- Operations – the number of operations (reads, inserts, updates or cleanups) performed during the benchmark;
- Avg. latency – as name says average latency during performing operations. Calculated in 0.001 milliseconds;
- Min. latency – the minimum achieved latency during the benchmark. Calculated in 0.001 milliseconds;
- Max. latency – the maximum achieved latency during the benchmark. Calculated in 0.001 milliseconds;
- 95^{th} percentile latency – latency bound for 95% of all operations. Calculated in milliseconds; and
- 99^{th} percentile latency – latency bound for 99% of all operations. Calculated in milliseconds.

For scalable RPM benchmarking we have instantiated one master and five region servers. Initially, we benchmark a single node cluster and compare these results with those obtained using MySQL running on a single machine and then we increased the number MySQL instances (the number of shards) as well as the number of nodes in the cluster. Therefore, we add one node to the cluster after performing each set of benchmarks. We expected that this would give us a clear picture of how many machines are necessary in the cluster to provide clinicians with the ability to observe the health measurements of patients in (near) real-time. After we achieve the expected performance with a given number of machines in the cluster, we increase the number of patients in the system (simulated by adding several millions of rows in the data stores) and then again performed benchmarks while adding machines until we get the expected performance. This allowed us to calculate how much patient data can be efficiently managed by one node and when we need to add additional nodes to the cluster (we need to know if all patients transmit their health data at certain times in a day, then for a given number of newly registered patients we need to add an additional node to meet the performance requirements of the system). These tests should allow us to characterize the performance of the system when the number of nodes is increased one by one.

The number of tests in each stage is three. Initially we perform 500,000 operations on one million rows of 1KB data, calculate throughput and latency, and then we perform ten million operations on ten million and fifty million rows of 1KB data respectively (again calculating throughput and latency). After each of these tests we save the benchmark results for later analysis. Additionally, we used several threads to make the benchmark process faster. The number of threads was different at each stage. Before starting the benchmarking process, we loaded our generated data into both MySQL and HBase. After each stage both storage systems were emptied and a new data set with a different amount of data was loaded. To obtain more precise results, before running workloadE both the MySQL and HBase databases were emptied, as prior workloads inserted additional data during their writing phase, hence the number of rows in the system was increased.

5.4.1 Machine specifications

All of the benchmarks were run on machines with identical specifications, see Table 5-2.

Table 5-2: Machine Specifications

Machine	Dell Inc. PowerEdge 2970
Processor	Hynix Semiconductor (Hyundai Electronics), AMD Opteron™ Processor 2435 with six cores clocked at 2592.994 MHz
Memory	8 x 4096 MB DDR2 with 800 MHz (1.2 ns) speed
Disk	Western Digital, 4 x 256 GB, disk cache size 512 KB, cached read speed 8544 Mbps, buffered disk read speed 81 Mbps
Mother board	Dell 0JKN8W
Disk controller	Dell SAS PowerEdge RAID Controller (PERC)
Network Interface	Embedded Broadcom 5708 NIC 1, NIC 2

All machines in the cluster (in case of multi-node cluster) are interconnected by an Intel® Gigabit Ethernet Switch with the network interfaces configured for 1 Gbps in full-duplex mode and the network interfaces performed checksum computations.

5.4.2 YCSB Benchmark on MySQL DB

The MySQL database is sharded several times while performing the benchmarks. Initially, the benchmark was performed with a single instance of MySQL and for consequent operations the number of shards was three, four, and five. For each stage of testing the desired amount of data was loaded and then the workloads are run one by one. The following command-line script loads one million rows of data into the MySQL database with ten threads and writes the loading statistics to the file '1-mysql-load.dat':

./bin/ycsb load jdbc -P workloads/workloada -P jdbc-binding/conf/db.properties -p recordcount=1000000 -p threadcount=10 -s | tee -a benchmark-results/1-mysql-load.dat

Table 5-3 presents the statistics obtained while loading different numbers of rows of data. This table shows the number of threads, throughput (in operations per second - ops/sec), and the range of latency values per operation.

Table 5-3: MySQL Data Load

Million rows	1	10	50
Elapsed Time (ms)	51,622.0	2,682,092.0	139,923,732.0
Thread Count	10	20	100
Throughput (ops/sec)	19,371.586	10,728.433	357.338
Average Latency (ms)	0.435	2.165	11.534
Minimum Latency (ms)	0	0.006	0.12
Maximum Latency (ms)	1,009.412	38,122.511	195,333.092
95th Percentile Latency (ms)	0	4	15
99th Percentile Latency (ms)	0	7	25

After loading data, we start the benchmarking process by running workloadA. By default, YCSB client uses a single worker thread, but also additional threads can be specified to increase the amount of load offered against the database which is what we need. For this reason, we picked ten threads* to perform our first operation and for subsequent operations we increased the number of worker threads. The following script performs 500,000 read and update operations on a table with one million rows using ten threads:

./bin/ycsb run jdbc -P workloads/workloada -P jdbc-binding/conf/db.properties -p recordcount=1000000 -p threadcount=10 -p operationcount=500000 –p table=vital_sign -s | tee -a benchmark-results/1-mysql-workloadA-run.dat

Table 5-4 shows the benchmark results obtained during different types of workloads. The number of rows in this benchmark is one million, the number of operations is 500,000, and ten threads concurrently perform operations.

Table 5-4: MySQL Data Benchmark with 1 Million Rows

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	49,023	36,982.0	39,444.0	156,638.0
Rows (Millions)	1	1	1	1
Operations (Millions)	0.5	0.5	0.5	0.5
Thread Count	10	10	10	10
Throughput (ops/sec)	10,199.294	13,520.09	12,676.199	3,192.073
Average Latency (ms) (read)	0.098	0.848	0.798	5.023
(write)	0.911	NA	0.024	0.019
Minimum Latency (ms) (read)	0	0	0	0.191
(write)	0	NA	0	0
Maximum Latency (ms) (read)	1,739.802	913.03	1,023.553	736.496
(write)	4,133.882	NA	1.992	4.904
95th Percentile Latency (ms) (read)	0	1	1	24
(write)	1	NA	0	0
99th Percentile Latency (ms) (read)	0	1	1	35
(write)	2	NA	0	0

The rest of the benchmark measurement results are given in Appendix B, **Error! Reference source not found.** and Table B.1-2.

* This number of worker threads was picked only to increase the load on the database.

As mentioned above, MySQL database is partitioned several times for the purpose to see the clear performance difference between HBase. Besides, it was also vital to observe how RDBMS offers efficiency against scalability. Below tables represent the numbers obtained from load phase of YCSB after increasing the number of MySQL shards. Specifically, the numbers of Table 5-5 are those for MySQL with four shards, while Figure 5-6 presents statistics for MySQL with five shards.

Table 5-5: MySQL Data Load after MySQL Shard Reconfiguration (Four instances)

Million Rows	50	100
Elapsed Time (ms)	5,340,441.0	13,291,283.0
Thread Count	20	20
Throughput (ops/sec)	9,362.523	7,523.728
Average Latency (ms)	16.412	19.002
Minimum Latency (ms)	0.02	0.053
Maximum Latency (ms)	64,423.982	74,912.412
95th Percentile Latency (ms)	39	46
99th Percentile Latency (ms)	54	67

Table 5-6: MySQL Data Load after MySQL Shard Reconfiguration (Five instances)

Million Rows	100	250
Elapsed Time (ms)	10,845,083.0	29,387,911.0
Thread Count	20	50
Throughput (ops/sec)	9,220.769	8,506.899
Average Latency (ms)	16.293	22.081
Minimum Latency (ms)	0.038	0.061
Maximum Latency (ms)	61,012.512	69,010.116
95th Percentile Latency (ms)	40	58
99th Percentile Latency (ms)	66	89

Like the above tables, the following tables show the statistics obtained when running workloadC (read-only) with several million operations on several million rows:

Table 5-7: MySQL Data Read (Workload C) after MySQL Shard Reconfiguration (Four instances)

Million Rows	50	100
Elapsed Time (ms)	2,705,748	16,772,404.0
Million Operations	10	50
Thread Count	20	20
Throughput (ops/sec)	3,695.834	2,981.087
Average Latency (ms)	3.444	5.228
Minimum Latency (ms)	0	0
Maximum Latency (ms)	14,012.992	24,792.885
95th Percentile Latency (ms)	11	18
99th Percentile Latency (ms)	19	27

Table 5-8: MySQL Data Read (Workload C) after MySQL Shard Reconfiguration (Five instances)

Million Rows	100	250
Elapsed Time (ms)	13,251,457.0	29,310,368.0
Million Operations	50	100
Thread Count	20	50
Throughput (ops/sec)	3,773.17	3,411.762
Average Latency (ms)	4.423	9.322
Minimum Latency (ms)	0	0.034
Maximum Latency (ms)	21,067.155	55,023.983
95th Percentile Latency (ms)	15	21
99th Percentile Latency (ms)	25	38

5.4.3 YCSB Benchmark on a Single Node Cluster

This is the first benchmark of our newly developed prototype and the current benchmark is run on a cluster consisting of a single machine. As we did earlier, initially the proper amount of data should be loaded to run the benchmark workloads. Table 5-9 shows the throughput and the latency values achieved by a cluster with a single machine. Following script is executed in the command-line to start the first loading process:

./bin/ycsb load hbase -P workloads/workloada –p table=vital_sign -p columnfamily=vital_signs -p recordcount=1000000 -p threadcount=10 -s | tee -a benchmark-results/1-hbase-workloadA-load.dat

Table 5-9: HBase Data Load on a Cluster of Single Machine

Million rows	1	10	50
Elapsed Time (ms)	109,916.0	1,695,569.0	151,923,732.0
Thread Count	10	20	100
Throughput (ops/sec)	9,097.857	5,897.725	329.112
Average Latency (ms)	0.957	3.318	14.872
Minimum Latency (ms)	0.005	0.005	0.12
Maximum Latency (ms)	12,701.866	51,209.271	107,534.235
95th Percentile Latency (ms)	0	0	4
99th Percentile Latency (ms)	0	0	5

Table 5-10 shows the read and write latency and throughput of the HBase data store when running on a single machine cluster. The benchmark parameters during this stage are the same as the first stage of the MySQL benchmark. The rest of the benchmark results run on a cluster of single and multiple machines are shown in Appendix B from Table B.1-1 to Table B.1-16. After each benchmark a new machine is added to the cluster and MySQL database is reconfigured accordingly.

Table 5-10: HBase Benchmark on a Cluster of Single Machine with 1 million Rows and 500,000 Operations

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	61,513.0	135,376.0	99,101.0	409,373.0
Rows (Millions)	1	1	1	
Operations (Millions)	0.5	0.5	0.5	0.5
Thread Count	10	10	10	10
Throughput (ops/sec)	8,128.363	3,693.417	5,045.358	1,221.38
Average Latency (ms) (read)	2.216	2.678	2.053	8.576
(write)	0.203	NA	0.028	0.047
Minimum Latency (ms) (read)	0.128	0.101	0.107	0.383
(write)	0	NA	0.006	0.008
Maximum Latency (ms) (read)	3,100.239	3,085.358	3,135.076	1,232.78
(write)	4,785.415	NA	3.349	10.962
95th Percentile Latency (ms) (read)	5	4	5	40
(write)	0	NA	0	0
99th Percentile Latency (ms) (read)	15	36	19	53
(write)	0	NA	0	0

5.4.4 YCSB Benchmark on a Multi Node Cluster

Our benchmark for the multi-node cluster is divided into two parts where the first part covers benchmark statistics before tuning HBase and the second part includes benchmark statistics after tuning HBase. The performance difference between these two parts is quite large. By default most of the features of HBase are disabled and administrators must enable them based on their specific needs. The following performance tuning operations were made *before* loading data into HBase [60]:

- Pre-created empty regions. By default only one region was created in HBase and all clients were writing to the same region until it is large enough to split and become distributed across the cluster;
- The auto flush value is set to false. By default it is set to true, which means every write to the data store was sent one at a time to the disk - which significantly reduces performance;
- The Write Ahead Log (WAL) is turned off. By default it is turned on, which means that a Region Server writes each put operation into WAL log. In our case, this is already logged at the application level, so this WAL is unnecessary for us; and
- JVM heap size is increased. By default the heap size is set to 1000 (megabytes) which is insufficient when managing a huge amount of data.

In this section we only provide benchmark results obtained *after* performance tuning of HBase. The benchmark results before tuning HBase can be found in Appendix B. Table 5-11 and Table 5-12 illustrates the benchmark results obtained during the loading phase and last two tables (Table 5-13 and Table 5-14) represent the statistics when running workloadC. As the key operation in scalable RPM is read/write, the load phase (which is write only) and workloadC (which is read only) are sufficient to compare the system's and to give use the data necessary to analyze the behavior of our prototype.

Table 5-11: HBase Data Load on a Cluster of Four Machines after HBase Tuning

Million Rows	50	100
Elapsed Time (ms)	2,241,166.0	4,814,933.0
Thread Count	20	20
Throughput (ops/sec)	22,309.815	20,768.721
Average Latency (ms)	2.884	4.481
Minimum Latency (ms)	0.013	0.021
Maximum Latency (ms)	50,121.821	63,252.173
95th Percentile Latency (ms)	0	1
99th Percentile Latency (ms)	0	2

Table 5-12: HBase Data Load on a Cluster of Five Machines

Million Rows	100	250
Elapsed Time (ms)	4,161,339.0	10,368,971.0
Thread Count	25	50
Throughput (ops/sec)	24,030.727	24,110.395
Average Latency (ms)	2.761	5.559
Minimum Latency (ms)	0.001	0.019
Maximum Latency (ms)	30,133.192	58,981.341
95th Percentile Latency (ms)	0	3
99th Percentile Latency (ms)	0	5

Table 5-13: HBase Data Read (Workload C) on a Cluster of Four Machines after HBase Tuning

Million Rows	50	100
Elapsed Time (ms)	5,693,667.0	31,251,932.0
Million Operations	10	50
Thread Count	20	20
Throughput (ops/sec)	1,756.337	1,599.901
Average Latency (ms)	11.093	14.888
Minimum Latency (ms)	0.621	0.76
Maximum Latency (ms)	69,423.092	78,523.633
95th Percentile Latency (ms)	28	35
99th Percentile Latency (ms)	99	126

Table 5-14: HBase Data Read (Workload C) on a Cluster of Five Machines

Million Rows	100	250
Elapsed Time (ms)	27,040,260.0	53,447,319.0
Million Operations	50	100
Thread Count	25	50
Throughput (ops/sec)	1,849.095	1,871.001
Average Latency (ms)	11.981	17.821
Minimum Latency (ms)	0.613	0.871
Maximum Latency (ms)	70,423.523	87,423.025
95th Percentile Latency (ms)	29	42
99th Percentile Latency (ms)	103	177

5.4.5 Fault-tolerance Benchmark of Scalable RPMS

To test the fault-tolerance of our prototype, we ran the scalable RPMS on a cluster of four nodes. Hadoop was configured to replicate the data on three different nodes. As for all of the other tests initially the data was loaded into the system. The amount of data loaded was 10 million rows x 1KB (for a total of 10 GB) and the number of operations is one million. Several seconds after running workloadA on the cluster four nodes, the first node was forcibly shut down* and later* it was brought back into operation and after 1.5-2 minutes another node

* The node was shutdown by executing a *$HADOOP_HOME/bin/hadoop-daemon.sh --config $HADOOP_HOME/conf/ stop datanode* command.

was shut down. During the each test run there were 5 simulated node failures. The main goal behind this test was to evaluate the fault-tolerance of our prototype. According to Hadoop's documentation, after each shut down, the master node should dynamically refer to the other nodes and continue to operate. During this benchmark process several nodes were shut down one by one. The benchmark process was successfully completed *without* throwing exceptions, which means that the master node properly managed the slaves despite the corresponding nodes suddenly being shut down.

Table 5-15 shows the benchmark statistics from this testing. From the table we can see that the elapsed time for the operations took more than when the same test was performed *without* any crashes during benchmarking (this data is shown in Table B.1-15 column workloadA). In total five nodes were failed one by one for around 55 seconds, thus for ~275 seconds three nodes performed the processing of the workload. Throughput and latency differences between these two sets of statistics are illustrated in Figure 5-1. As shown in the figure, even a three node cluster performed better than a four node cluster when the cluster experienced individual node crashes. It takes some time for a master node to distribute part of the load to the other slave nodes when one slave node suddenly crashes. Part of the decrease in performance is due to the fact that when a slave node returns to operation it has to be updated from the master node, thus reducing the service rate of the master node to external operations.

Table 5-15: Fault-tolerance benchmark with workloadA on a Cluster of Four Machines

	Elapsed Time (ms)	Row Count	Operations	Thread Count	Throughput (ops/sec)	Average Latency (ms)
Workload A	1,102,512	10,000,000	1,000,000	20	907.02	127.52 (read) 1.19 (write)

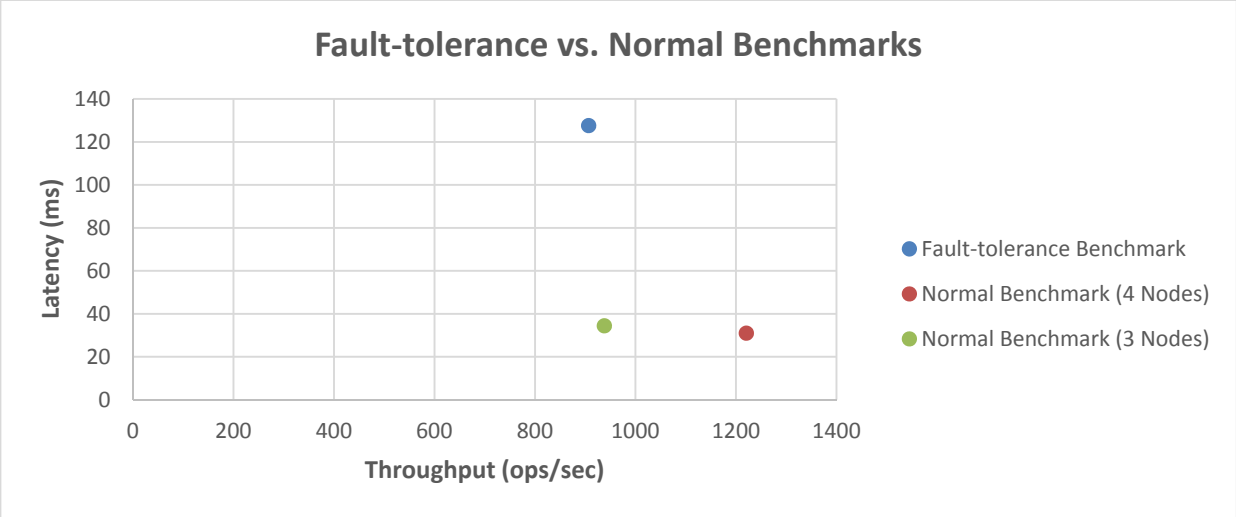


Figure 5-1: HBase benchmark differences between system with repeated machine failures and a system *without* such failures

* 50-60 seconds later

5.5 Benchmark Analysis and Comparison

In this section we collect all of the benchmark results in order to compare them with each-other and to extract meaningful information from these results. As mentioned above, the data load phase was performed at the beginning of each benchmark. Initially we analyze benchmark results of sharded MySQL and HBase before tuning and then later we analyze the systems after performance tuning.

Based on Table 5-3, Table 5-9, Table B.1-5, Table B.1-9, and Table B.1-13, Figure 5-2 depicts the load latency versus throughput of the relevant data stores. For initial benchmarks, the MySQL database was horizontally partitioned into three shards and this number of shards was kept for several benchmarks. Figure B.2-1 and B.2-2 show the case for 10 GB (10 million rows) and 50 GB (50 million rows) data loads respectively. In both cases, the performance of the two HBase machines exceeded the performance of the MySQL implementation.

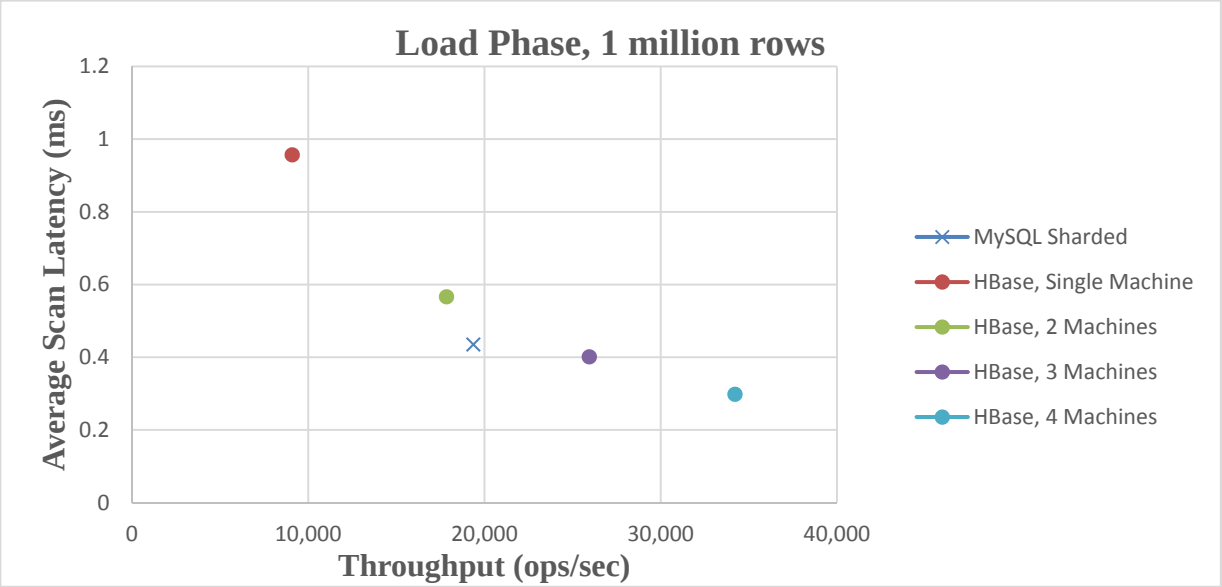


Figure 5-2: 1 GB of Data Load Benchmark Statistics

After loading the desired amount of data, the MySQL benchmark transaction phase was performed and subsequently the HBase benchmark was carried out. In the loading phase, sharded MySQL showed better performance than our prototype when running on a cluster with a single machine. However, as was shown in Figure 5-2, MySQL with three shards was out performed by HBase when the number of nodes in the cluster is greater than two. Additionally, the benchmarks show that the write operation in HBase is much faster than the write operation in MySQL.

Figure 5-3 illustrates workloadA's read latency of both the MySQL and HBase databases. The figure shows that MySQL has the lowest latency among all other benchmarks, because MySQL's read operation is much faster than a read operation in HBase. Accordingly, Figure 5-4 shows the write latency of MySQL and HBase when running the same workload. This figure illustrates that the write latency of MySQL is higher than the write latency of even a single node HBase cluster.

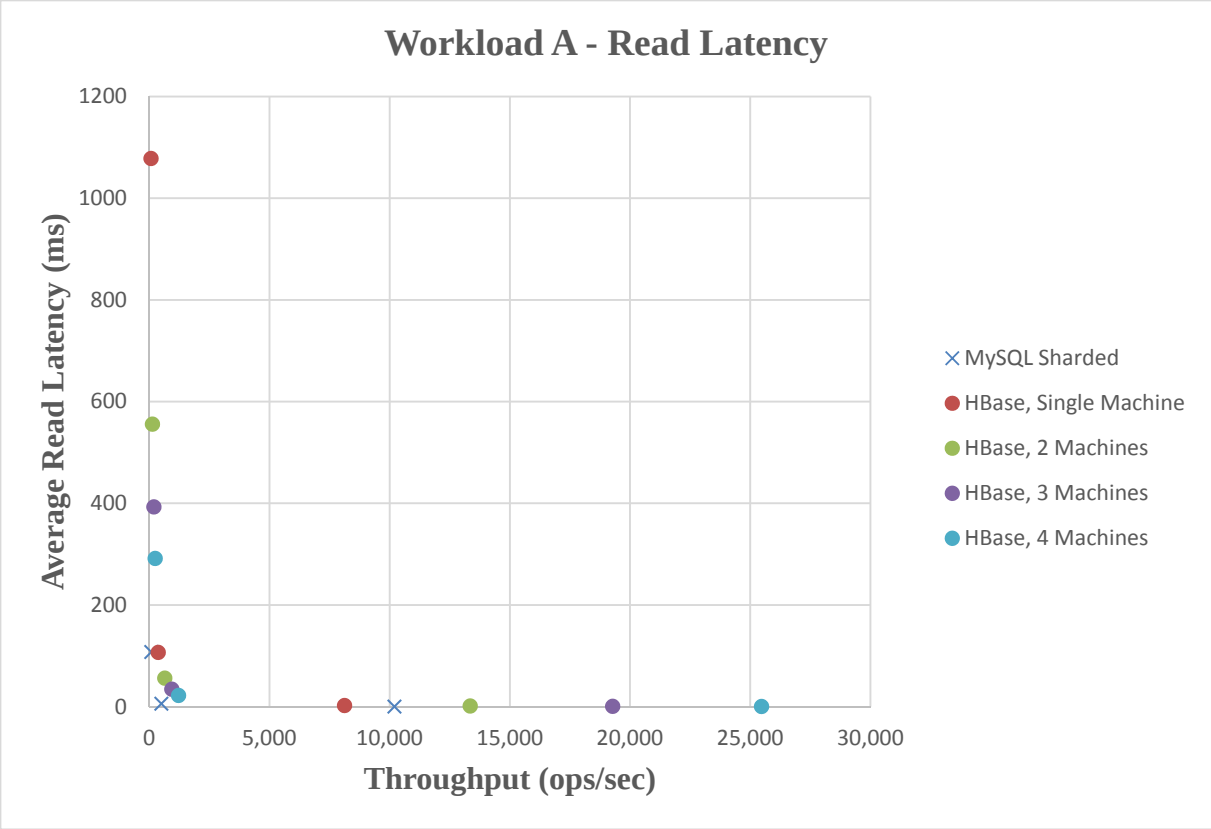


Figure 5-3: WorkloadA Read Latency vs. Throughput Benchmark Statistics

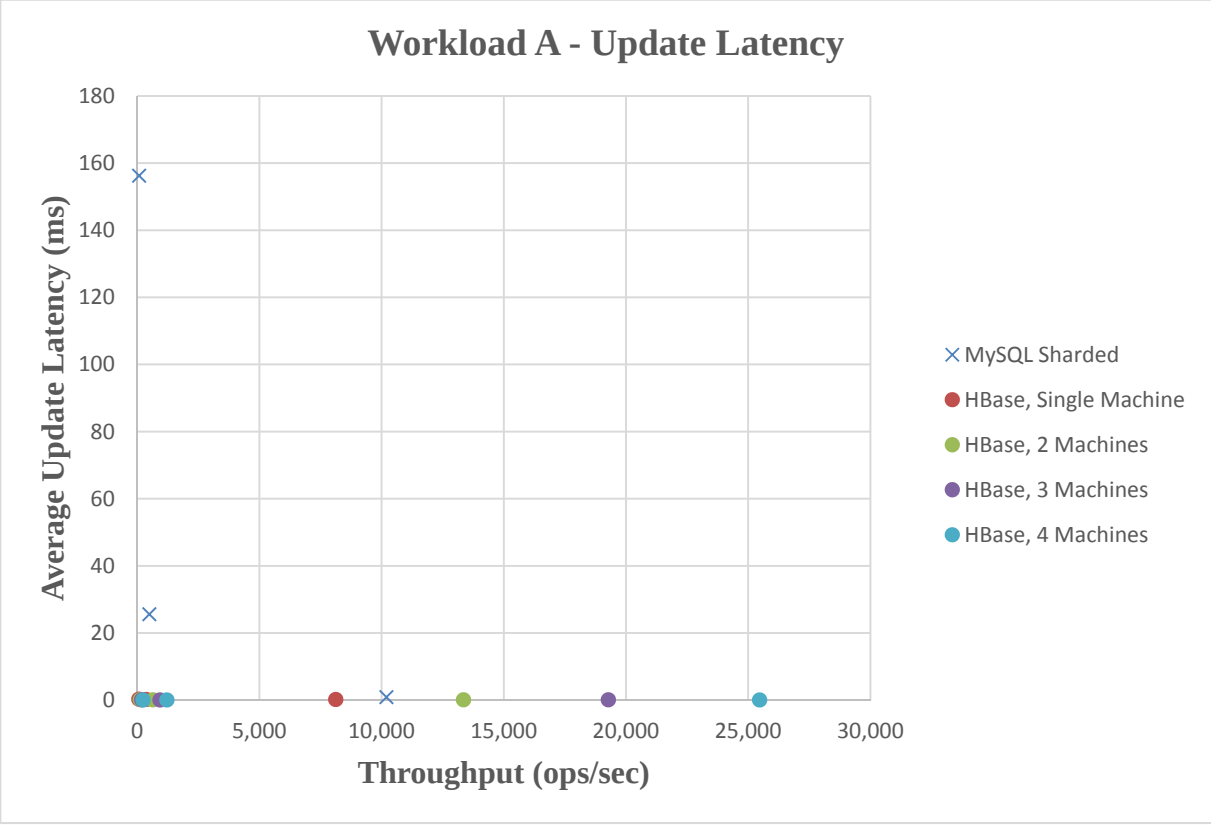


Figure 5-4: WorkloadA Write Latency Benchmark Statistics

Figure 5-5 illustrates the same benchmark results with regards to elapsed time and the number of operations. As shown in this figure, the elapsed time for MySQL to run workloadA is greater than the time of the HBase running on a cluster with a single machine. However, after adding one node to the cluster, HBase starts to show better performance than MySQL.

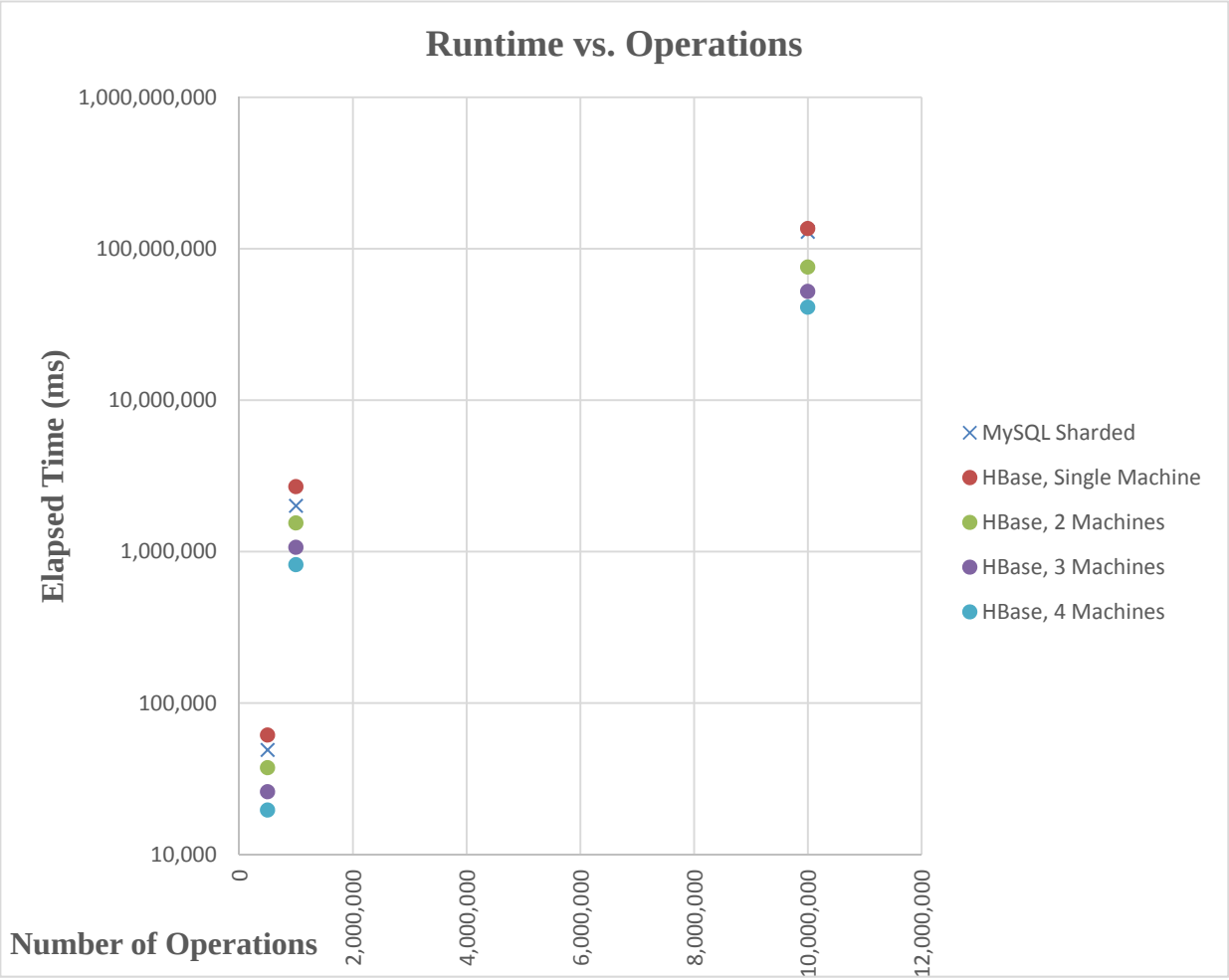


Figure 5-5: Elapsed Time (runtime) versus Number of Operations for WorkloadA

Further plots of the results of the benchmarking are given in Appendix B Figure B.2-1 to Figure B.2-10. Specifically, Figure B.2-3 illustrates read latency and throughput comparisons obtained running workloadC. The results show the clear difference between a traditional RDBMS and NoSQL. As noticed in earlier benchmarks, read operations in HBase are more expensive than a write operation, while in MySQL the reverse is true. Therefore, the performance difference between the two data stores increases with increasing numbers of write operations. In the first stage of the workloadC benchmark when the number of operations performed is 500,000, the single machine MySQL configuration showed better performance than a cluster of three machines; however, when the number of nodes in the cluster increased to four, then HBase started to perform better. Details of this benchmark results can be seen in Figure B.2-3 and Figure B.2-4.

Figure B.2-5 – Figure B.2-10 characterize comparisons between MySQL and HBase cluster running workloadD and workloadE respectively. In both of them MySQL falls behind when the number of nodes in the cluster is more than one. In the benchmarks with fewer less operations and less data, MySQL performed much better than HBase running on a cluster with a single machine. However, when the number of operations increases or the amount of

data increases, then the performance of MySQL starts to decrease much more quicker than HBase's performance.

Both MySQL and HBase performed poorly when the number of operations reached 1 million or more. Even a cluster with four nodes spent much time to perform operations. However, after performance tuning we started to achieve more acceptable results.

Figure 5-6 shows 100 million rows of data load on both HBase and MySQL data stores. As shown in the figure, the performance after tuning is much higher. The figure clearly illustrates that MySQL has much slower performance when it comes to data loading (i.e., write operations).

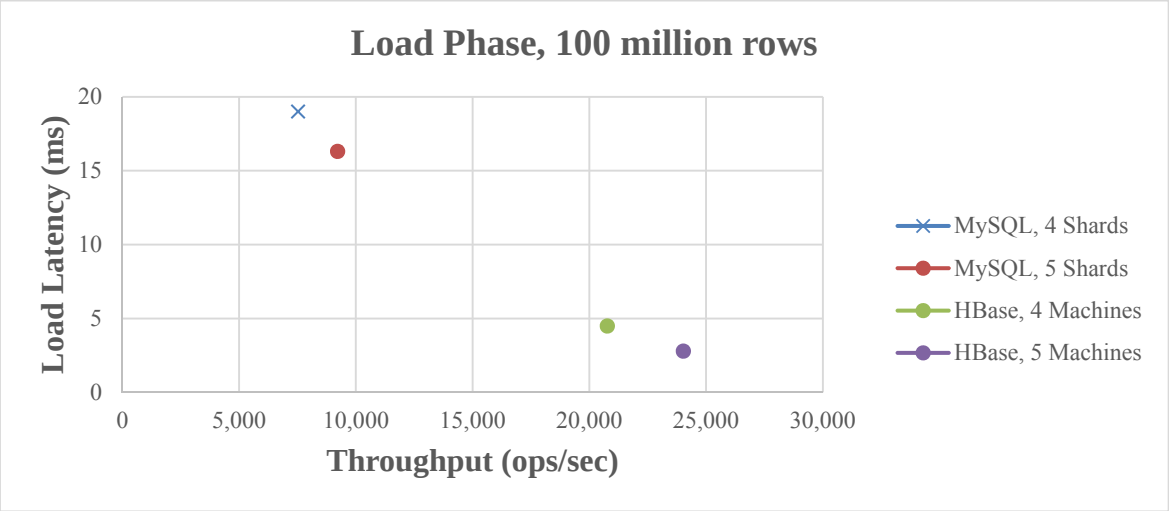


Figure 5-6: Data Load Benchmark Statistics for 100 GB

Subsequently, the number of rows is raised to 250 million and the benchmark is performed again. Figure 5-7 depicts 250 million lines of row load on MySQL and HBase. These results clearly indicate that the throughput of HBase is far more ahead of MySQL during performing load operations. From these values, we can assume that HBase cluster with five machines can achieve throughputs ranging from 20 to 26 thousand operations per seconds.

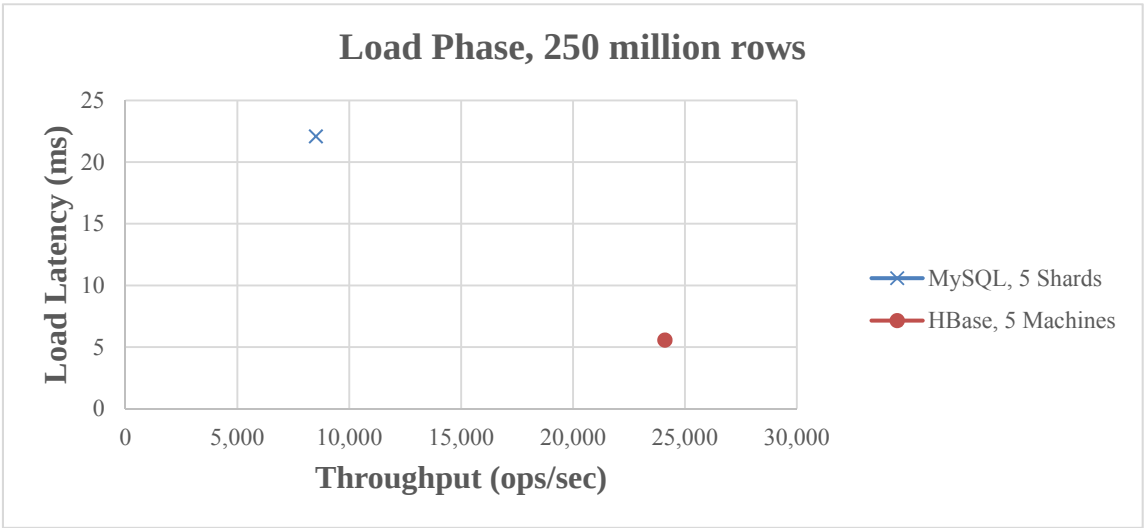


Figure 5-7: Data Load Benchmark Statistics for 250 GB

Figure 5-8 and Figure 5-9 represent workloadC benchmark results with 50 million and 100 million operations respectively. As shown in the figures, MySQL performed better than HBase which means that the read operation in MySQL is faster than in HBase.

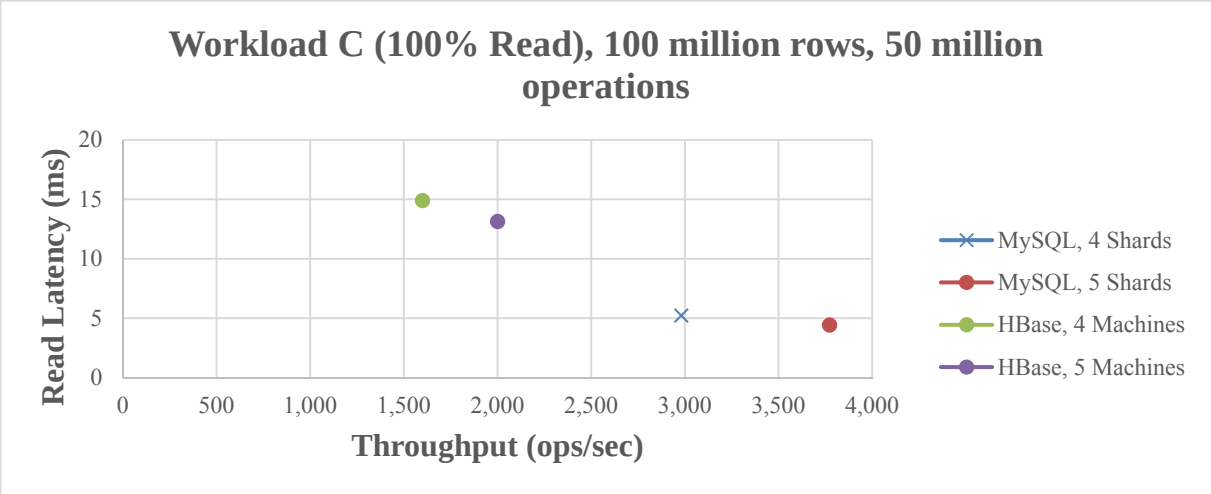


Figure 5-8: WorkloadC Benchmark Statistics on 100 million rows

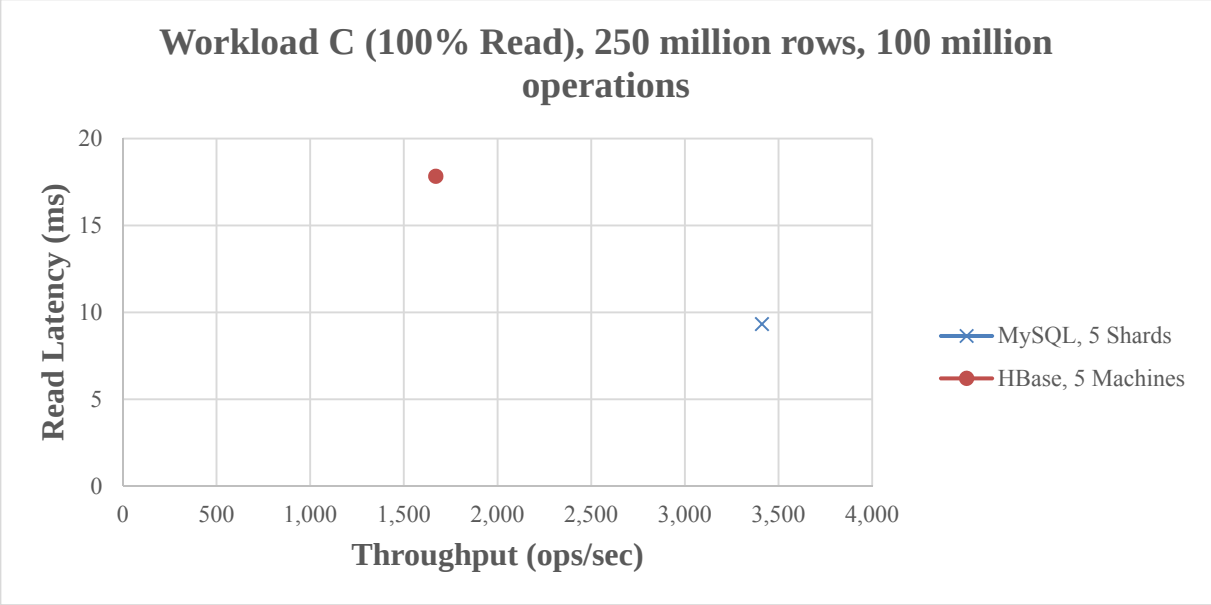


Figure 5-9: WorkloadC Benchmark Statistics on 250 million rows

Another important set of values in our benchmark are the 95th and 99th percentile latencies. These show the latency bound for 95% and 99% of the operations. We show this data as the average latency values may not indicate whether the system performs as expected, because according to our benchmark results, the difference between the average latency and these two bounds are quite large. Specifically, as the number of operations is increased the 95th and 99th percentile latencies are higher than average latency, this means that the average is being affected by the extremely small latencies of a large fraction of the operations. Figure 5-10 presents the differences among these three read latency values. As the figure shows, the average latency values are lower than the 95th and 99th percentile latencies.

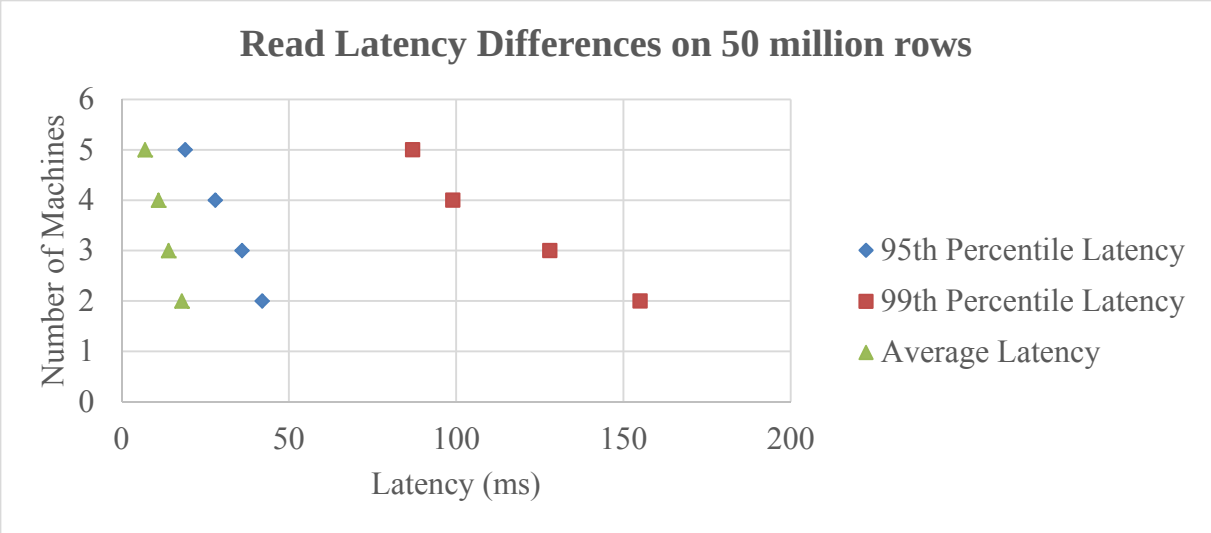


Figure 5-10: HBase Read Latency Differences on 50 million rows

In order to request patient data clinicians' requests are entered into a queue in the RPMS and one by one the queries are processed to retrieve the relevant health records. We can calculate the maximum number of requests to accessing a database containing 250 million rows with a maximum desired latency of 2 seconds. We used the benchmark statistics obtained running workloadD, because we need a workload which performs both reads and writes. Table 5-16 represents the benchmark statistics obtained by running workloadD on a cluster of five machines with 250 million rows.

Table 5-16: HBase Data Read/Write (Workload D) on a Cluster of Five Machines

Million Rows	Elapsed Time (ms)	Throughput (ops/sec)	Average Latency (ms)	95th Percentile Latency (ms)	99th Percentile Latency (ms)
250	49,634,829.0	2,014.714	14.346	40	138

From these benchmark results we extract those that we need to determine the maximum number of requests which could be satisfied in the bounded time. Let us assume that the average throughput on a five machine cluster performing workloadD is ~2,000 operations per second, then a maximum of 4,000 clinicians can make a query about a patient and have a 2 seconds latency bound if each of these clinicians makes one query at a time and all of them make their queries at the same time and all of them make a new query every two seconds. If we assume that a clinician makes less than one query per second and that each query requires some number of operations then we can apply Little's law [61] to our results from both HBase and MySQL in order to calculate the maximum number of clinicians who could be satisfied by a system which runs sharded MySQL or HBase as a database storage. It is obvious that MySQL is the best choice with regard to satisfying queries since the number of read queries that can be satisfied per unit time is larger than for HBase.

Next we calculate the scalability of our prototype. Based on Figure 5-6 and Figure 5-8, we can compute how rapidly we can scale out our prototype. We utilized Amdahl's law [62] for our analysis of scaling. Amdahl's law says that the speedup of an application using multiple processors in parallel is limited by the time needed for the sequential fraction of the application. In other words, the law states that if each application has one part that takes *t* time to be processed and it is impossible to execute this part in parallel, then regardless of how

many processors are devoted to parallelize the execution of this application, the minimum execution time cannot be less than *t*. A formula for the maximum speedup, *S(n)*, is:

$$S(n) = \frac{1}{r_s + \frac{r_p}{n}} \quad (1)$$

where, $r_s + r_p = 1$, r_p represents the portion of the application that can be occured in parallel, r_s represents the sequential portion of the application, and n is the number of machines (processors).

To analyze the scaling of our prototype, first we need to find r_s. To calculate it, we need to know the elapsed time difference between two sets of machines running the same workload. We took runtimes of data reads (workloadC) phase with four and five nodes (represented in Table 5-11 and Table 5-12) and the time difference is as follows:

Difference = t_5 / t_4 = 27,040,260 / 31,251,932 = ~0.865

In ideal case, the difference would satisfy the following equation:

$$n \times t_n = (n+1) \times t_{n+1} \quad => \quad \frac{n}{n+1} = \frac{t_{n+1}}{t_n}$$

where, *n* – number of machines and *t* – elapsed time for the operation to be finished. From this equation, we obtain 4 / 5 = 0.8. Now it is obvious that $r_s = 0.865 - 0.8 = 0.065 = 6.5\%$. So 6.5% of our application was sequential. From this value, we calculate r_p which is $r_p = 1 - r_s = 0.935 = 93.5\%$.

Based on (1) equation, we calculate S(5) which equals to 3.97 and S(4) = 3.35. We also calculated the case of a three machine cluster and the sequential portion was $r_s = 6.2\% = \sim 6\%$ which led us to gain S(3) = 2.68.

These last calculations clearly show the speedup of our application. From those values we conclude that only ~94% of our application can be parallelized while the remaining portion (~6%) will not be processed in parallel. Equation (1) shows that even if the number of machines were infinite, the maximum speedup will be S(∞) = 1 / 0.06 = 16.67. From this we learn that by the time we have scaled up to 23 machines we are getting only a small gain in speedup when adding an additional machine.

6 Conclusions and Future Work

This chapter concludes the thesis project providing a conclusion and suggesting additional technologies to be applied in future research. Final section of this chapter includes some required reflections on social, economic, and ethical considerations.

6.1 Conclusions

The primary goal of this thesis was to design a new prototype of a highly efficient scalable remote patient monitoring system which delivers near real-time health measurements of patients to clinicians and also to compare the new system against traditional systems which utilize relational databases. As we planned in the beginning of our thesis project, the main attention was paid only to the realization of the first part of the RPM system where patients transfer their health measurements and physicians monitor them in ordinary manner. Mining obtained medical records and advanced patient monitoring (mainly for patients whose health state is below than normal) was outside the scope of this master's thesis. In Figure 6-1, the area edged with blue lines shows the parts of a modern RPMS that were developed by us during the period of this thesis project. Consequently after the development, a number of benchmarks were performed in order to prove that the prototype delivers near real-time measurements for analysis. The statistical analysis of these benchmark results showed the performance difference between relational and non-relational databases when performing write, read, and scan operations. Additionally, the fault-tolerance of the developed prototype was evaluated using benchmarks. All of the benchmark results are presented in Chapter 5 and Appendix B.

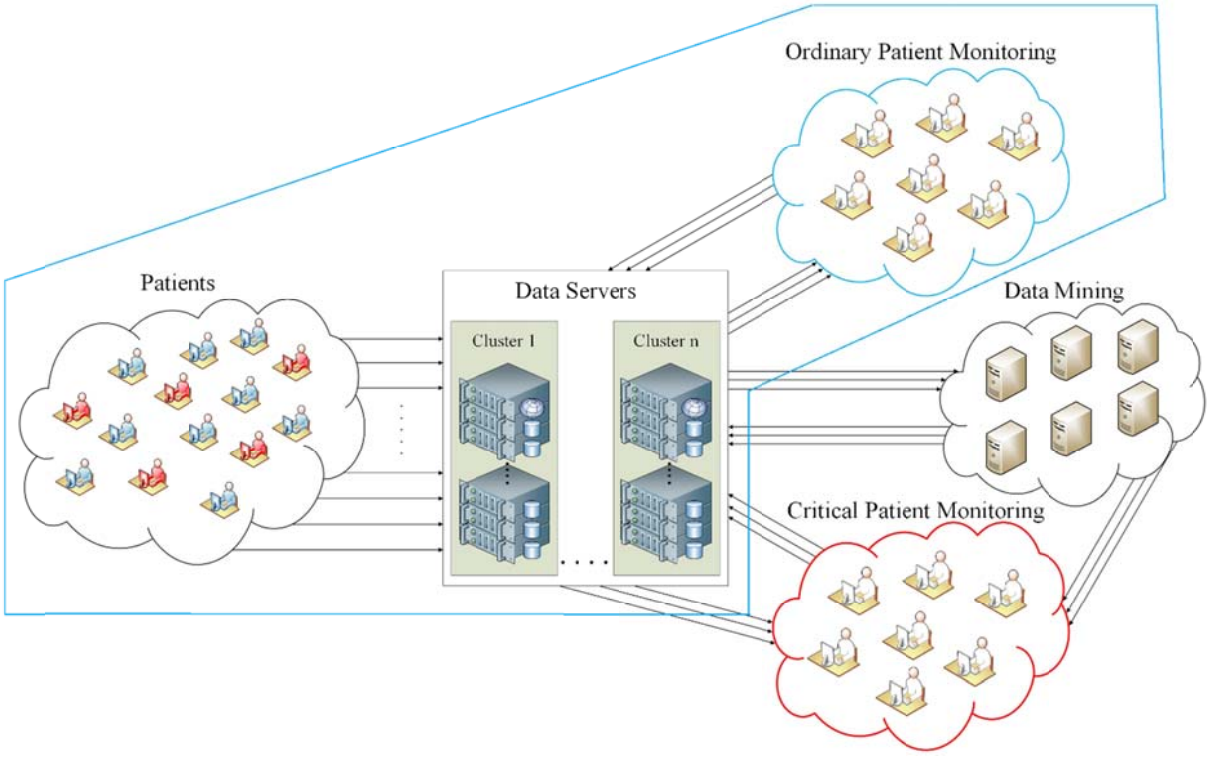


Figure 6-1: Modern RPMS Architecture

In the prototype Apache HBase is used to store health measurements of patients and therefore the performance of storing and retrieving such records was measured by running benchmarks. In order to contrast the performance of HBase, the MySQL relational database was chosen as it represents one of the most widely used RDBMS. Benchmark statistics showed that HBase has unquestionable superiority in terms of write performance, while MySQL performed much better in terms of read performance. In terms of executing complex queries for medical record statistics, both data stores performed poorly - although HBase performed better than MySQL.

It is important to note that chosen data store for our prototype, HBase, showed both satisfactory and unsatisfactory performance results during executing workloads. Benchmark tests also include two stages: before and after performance tuning of HBase. The performance difference after tuning was much higher than before. Another factor which is worth mentioning is the slowness of read operation in HBase as compared to MySQL, because the read operation is as critical as write operation in RPMSs. However, our system does not include critical patient monitoring which requires significantly high speed data reads. Additionally, the write rate is higher than the reading rate. Because when a patient's health state is as expected, physicians usually request health measurement statistics for the last week or the last two weeks, etc. Considering these factors we assert that HBase is better option; however when it comes to provide a high fraction of data reads, MySQL is the most suitable candidate.

6.2 Future Work

As described in Section 1.2.1, current master's thesis proposed and designed a system which is considered as the first part of today's modern RPMSs and the rest two parts have to be developed as a future work of this project. More in detail, Figure 6-1 represents modern RPMS with all three parts where 'Data Mining' and 'Critical Patient Monitoring' parts dedicated as a future work. The health care for different kinds of patients has to be provided accordingly; hence intelligent analysis of measurement results is highly important for clinicians as well as patients.

Another crucial functionality that should be implemented in RPMS is specific data searching using Solr. Since, Solr provides ultra-fast search services while keeping a simple programming model, its implementation in RPMS would be invaluable. Because especially, in the third part of RPMS, clinicians frequently request health statistics of specific patients for the last several days or weeks or even months and it is vital that the query should not take long regardless of its complexity.

Providing security in a scalable RPMS was outside the scope of this master's thesis project. As most medical records need to be securely transmitted and managed, enhanced security is essential to such a system in the real-world. Specifically, authentication and authorization mechanisms have to be made to function properly in Hadoop. User identification (authentication) can be done in many ways, including making changes in low-level transport, using the Kerberos authentication protocol or token delegation among users. Authorization should include all types of access controls to resources and role-based management.

Today the majority of current remote patient monitoring systems use traditional relational databases to store and manage their data. Highly scalable data management techniques were introduced only a few years ago due to the need to handle big data. We expect that increasingly there will be an integration of both scalable and relational data management technologies in RPMS. One of the best ways to integrate Hadoop with RDBMS is using

Apache Sqoop. Utilizing this data integration tool in a scalable RPMS would make the system even more extensible and capable.

6.3 Reflections

This master's thesis project facilitates patient monitoring which helps both patients and clinicians. The full implementation of this project in the real-life could help to avoid long queues in hospitals, facilitate patients communicating with their clinicians, and enable clinicians to efficient access patients' health records. Increasing the effectiveness of the relation between patients and physicians is considered as a beneficial social aspect of this master's thesis project. The key economic aspect of this thesis project is its cost efficiency, as it describes a method that can be used to provide a scalable RPMS which can support very large amounts of medical records – especially those records which will be coming from home health care monitoring devices. An environmental aspect of this work is a potential reduction in need for emergency transport (by helicopter or ambulance) due to better observation of patient's conditions.

References

1. Nicole Lewis, “Remote Patient Monitoring Market To Double by 2016,” Jan-2012. [Online]. Available at: http://www.informationweek.com/healthcare/mobile-wireless/remote-patient-monitoring-market-to-doub/240004291. [Accessed: 21-Feb-2013].
2. Ken Terry, “Remote Patient Monitoring Shows Strong Growth,” Jan-2012. [Online]. Available at: http://www.informationweek.com/healthcare/mobile-wireless/remote-patient-monitoring-shows-strong-g/232301359. [Accessed: 21-Feb-2013].
3. Berg Insight, “Berg Insight says 2.8 million patient are remotely monitored today,” Jan-2013. [Online]. Available at: http://www.berginsight.com/News.aspx?m_m=6&s_m=1. [Accessed: 21-Feb-2013].
4. Iain Morris, “Remote patient monitoring systems to grow to 9.4 million by 2017: Berg Insight,” Jan-2013. [Online]. Available at: http://www.telecomengine.com/article/remote-patient-monitoring-systems-grow-94-million-2017-berg-insight. [Accessed: 21-Feb-2013].
5. Center for Aging and Technology, “Technologies for Remote Patient Monitoring for Older Adults,” An Initiative of The SCAN Foundation and Public Health Institute, Apr-2010. [Online]. Available at: http://www.techandaging.org/RPMPositionPaper.pdf. [Accessed: 22-Mar-2013].
6. Pete Larson, “Remote Patient Monitoring for Home Health,” HEALTH Interlink, Jun-2012. [Online]. Available at: http://healthinterlink.com/images/HealthInterlink_HH_Webinar2.pdf. [Accessed: 22-Mar-2013].
7. An Oracle White Paper, “Oracle: Big Data for the Enterprise,” Jan-2012. [Online]. Available at: http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf. [Accessed: 21-Feb-2013].
8. April Reeve, “Big Data and NoSQL: The Problem with Relational Databases,” Sep-2012. [Online]. Available at: http://infocus.emc.com/april_reeve/big-data-and-nosql-the-problem-with-relational-databases/. [Accessed: 21-Feb-2013].
9. Paul C. Zikopoulos, Chris Eaton, Dirk deRoos, Thomas Deutsch, and George Lapis, “Understanding Big Data,” IBM Corporation, McGraw-Hill, 2012. ISBN 978-0-07-179053-6, pp. 3–13. Available at: http://public.dhe.ibm.com/common/ssi/ecm/en/iml14296usen/IML14296USEN.PDF. [Accessed: 22-Feb-2013].
10. IBM Corporation, “Bringing Big Data to the Enterprise: What is Big Data,” Jan-2013. [Online]. Available at: http://www-01.ibm.com/software/data/bigdata/. [Accessed: 22-Feb-2013].
11. Dion Hinchcliffe, “10 Ways to Complement the Enterprise RDBMS using Hadoop,” Sep-2007. [Online]. Available at: http://www.ebizq.net/blogs/enterprise/2009/09/10_ways_to_complement_the_ente.php. [Accessed: 16-Mar-2013].
12. Hortonworks, “Understanding Hadoop Ecosystem.” [Online]. Available: http://docs.hortonworks.com/CURRENT/index.htm#About_Hortonworks_Data_Platform/Understanding_Hadoop_Ecosystem.htm. [Accessed: 22-Feb-2013].
13. Deborah Lee Soltesz, “The Advantages of a Relational Database Management System.” [Online]. Available at: http://www.ehow.com/list_6121487_advantages-relational-database-management-system.html. [Accessed: 26-Feb-2013].
14. Anni Martin, “Disadvantages of Relational Database.” [Online]. Available at: http://www.ehow.com/list_5977286_disadvantages-relational-database.html. [Accessed: 26-Feb-2013].
15. Bart Jacob, Michael Brown, Kentaro Fukui, and Nihar Trivedi, “Introduction to Grid Computing,” IBM Corporation, Dec-2005. pp. 3-17. [Online]. Available at: http://www.redbooks.ibm.com/redbooks/pdfs/sg246778.pdf. [Accessed: 12-Mar-2013].
16. Heli Xu and Guixin Wu, “Parallel Programming in Grid: Using MPI,” presented at the Proceedings of the Third International Symposium on Electronic Commerce and Security Workshops, Jul-2010. ISBN 978-952-5726-11-4. pp. 136–138. [Online]. Available at: http://www.academypublisher.com/proc/isecs10w/papers/isecs10wp136.pdf. [Accessed: 14-Mar-2013].

17. Luis F. G. Sarmenta, “Volunteer Computing,” MIT, Department of Electrical Engineering and Computer Science, Jun-2001. [Online]. Available at: http://www.dmut.net/en/282/282078.pdf. [Accessed: 13-Mar-2013].
18. David P. Anderson, “BOINC: A System for Public-Resource Computing and Storage,” Nov-2005. [Online]. Available at: http://www.cs.umd.edu/class/fall2005/cmsc714/Lectures/koren-boinc.pdf. [Accessed: 14-Mar-2013].
19. J2EEBrain, “Hadoop - Advantages and Disadvantages,” 2012. [Online]. Available at: http://www.j2eebrain.com/java-J2ee-hadoop-advantages-and-disadvantages.html. [Accessed: 26-Feb-2013].
20. Zak Stone, “Introduction to Hadoop,” 2011. [Online]. Available at: http://www.cs264.org/lectures/files/cs_264_hadoop_lecture_2011.pdf. [Accessed: 26-Feb-2013].
21. Carenet Team, “Carenet Services.” Jan-2013. [Online]. Available at: http://ttaportal.org/wp-content/uploads/2013/01/Final-Report-V-1.0.pdf. [Accessed: 23-Mar-2013].
22. Carenet Team, “Carenet Services: Sensor Desktop Application.” Jan-2013. [Online]. Available at: http://ttaportal.org/wp-content/uploads/2013/01/Sensors-Desktop-Application-v-1.0.pdf. [Accessed: 06-Mar-2013].
23. Carenet Team, “Carenet Services: Sensor Web Application.” Jan-2013. [Online]. Available at: http://ttaportal.org/wp-content/uploads/2013/01/Sensors-Web-Application-v-1.0.pdf. [Accessed: 21-Mar-2013].
24. Carenet Team, “Carenet Services: HDVC.” Jan-2013. [Online]. Available at: http://ttaportal.org/wp-content/uploads/2013/01/Sensors-Web-Application-v-1.0.pdf. [Accessed: 22-Mar-2013].
25. Fernández Alexis Martínez, “Authorization schema for electronic health-care records: For Uganda,” KTH, School of Information and Communication Technology (ICT), TRITA-ICT-EX-2012:176, Aug-2012. [Online]. Available at: http://kth.diva-portal.org/smash/record.jsf?pid=diva2:546619. [Accessed: 26-Mar-2013].
26. Sherin Sebastian, Neethu Rachel Jacob, Yedu Manmadhan, V. R. Anand, and M. J. Jayashree, “Remote Patient Monitoring System,” presented at the International Journal of Distributed and Parallel Systems (IJDPS), Sep-2012, pp. 99–110. DOI : 10.5121/ijdps.2012.3509. [Online]. Available at: http://airccse.org/journal/ijdps/papers/3512ijdps09.pdf. [Accessed: 26-Mar-2013].
27. Myung-kyung Suh, Chien-An Chen, Jonathan Woodbridge, Michael Kai Tu, Jung In Kim, Ani Nahapetian, Lorraine S. Evangelista, and Majid Sarrafzadeh, “A Remote Patient Monitoring System for Congestive Heart Failure,” presented at the Springer Science+Business Media, May-2011. DOI 10.1007/s10916-011-9733-y. [Online]. Available at: http://www.chime.ucla.edu/Evangelista-%20A%20Remote%20Patient%20Monitoring.pdf. [Accessed: 28-Mar-2013].
28. Myung-kyung Suh, Lorraine S. Evangelista, Victor Chen, Wen-Sao Hong, Jamie Macbeth, Ani Nahapetian, Florence-Joy Figueras, and Majid Sarrafzadeh, “WANDA B.: Weight and Activity with Blood Pressure Monitoring System for Heart Failure Patients,” University of California, Los Angeles, 2010. DOI: 10.1109/WOWMOM.2010.5534983, [Online]. Available at: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3075586/. [Accessed: 30-Mar-2013].
29. Alex Holmes, *Hadoop in Practice*. Shelter Island, NY 11964: Manning Publications Co., 2012. ISBN 9781617290237. [Accessed: 03-Apr-2013].
30. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System,” presented at the SOSP’03, Bolton Landing, New York, USA, Oct-2003. [Online]. Available at: http://www.cs.rochester.edu/meetings/sosp2003/papers/p125-ghemawat.pdf. [Accessed: 01-Apr-2013].
31. Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” presented at the OSDI ’04: 6th Symposium on Operating Systems Design and Implementation, 2004, pp. 137–149. [Online]. Available at: http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf. [Accessed: 01-Apr-2013].
32. Tom White, *Hadoop: The Definitive Guide, Second Edition*. 1005 Gravenstein Highway North: Sebastopol, 2010. [Accessed: 01-Apr-2013].
33. The Apache Software Foundation, “Hadoop Tutorial and Documentation,” Feb-2013. [Online]. Available at: http://hadoop.apache.org/docs/r1.0.4/. [Accessed: 05-Apr-2013].

34. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy, "Hive - A Warehousing Solution Over a Map-Reduce Framework," presented at the VLDB Endowment, Facebook Data Infrastructure Team, Lyon, France, Aug-2009. [Online]. Available at: http://www.vldb.org/pvldb/2/vldb09-938.pdf. [Accessed: 05-Apr-2013].
35. Alan Gates, *Data flow scripting with Hadoop: Programming Pig*, First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Oct-2011. [Accessed: 06-Apr-2013].
36. Nick Dimiduk and Amandeep Khurana, *HBase in Action*. Manning Publications Co., 2013. pp. 3-20. [Online]. Available at: http://www.manning.com/dimidukkhurana/HBiAsample_ch1.pdf. [Accessed: 06-Apr-2013].
37. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," presented at the OSDI, 2006. [Online]. Available at: http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/bigtable-osdi06.pdf. [Accessed: 06-Apr-2013].
38. Alan Gates, "Apache Hadoop* Community Spotlight: HCatalog." Aug-2012. [Online]. Available at: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/hadoop-spotlight-hcatalog-paper.pdf. [Accessed: 04-Jul-2013].
39. The Apache Software Foundation, "ZooKeeper." 2008. [Online]. Available at: http://zookeeper.apache.org/doc/r3.2.2/zookeeperOver.pdf. [Accessed: 27-June-2013].
40. Apache Software Foundation, "Apache Ambari." Apr-2013. [Online]. Available at: http://incubator.apache.org/ambari/. [Accessed: 09-Apr-2013].
41. John Speidel, "Ambari API Reference." Feb-2013. [Online]. Available at: https://github.com/apache/ambari/blob/trunk/ambari-server/docs/api/v1/index.md. [Accessed: 09-Apr-2013].
42. Apache Software Foundation, "Apache Sqoop," Mar-2013. [Online]. Available at: http://sqoop.apache.org/. [Accessed: 09-Apr-2013].
43. The Apache Software Foundation, "Apache Solr," 2012. [Online]. Available at: http://lucene.apache.org/solr/. [Accessed: 10-Jun-2013].
44. Eric Sammer, *Hadoop Operations: A Guide for Developers and Administrators*, First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Sep-2012.
45. Lars George, "Hive vs Pig," Oct-2009. [Online]. Available at: http://www.larsgeorge.com/2009/10/hive-vs-pig.html. [Accessed: 16-Apr-2013].
46. Lars George, *HBase: The Definitive Guide*. 1005 Gravenstein Highway North, Sebartol, CA 95472.: O'Reilly Media, Inc., Sep-2011.
47. Ruslan Mukhammadov, *Scalable Remote Patient Monitoring System App. Source code,* Jun-2013. [Online]. Available at: https://github.com/ruslanm/scalable-rpm. [Access: 08-Jul-2013].
48. Michael G. Noll, "Running Hadoop on Ubuntu Linux (Single-Node Cluster)," Mar-2013. [Online]. Available: http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/. [Accessed: 12-Jun-2013].
49. Michael G. Noll, "Running Hadoop on Ubuntu Linux (Multi-Node Cluster)," Apr-2013. [Online]. Available: http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/. [Accessed: 15-Jun-2013].
50. MapR Technologies, "Installing Apache HBase Standalone mode in Ubuntu," Jun-2013. [Online]. Available: http://www.mapr.com/doc/display/MapR/HBase. [Accessed: 12-Jun-2013].
51. Robert Chen, "Setup Multinodes HBase/Hadoop/Zookeeper on Ubuntu," Sep-2012. [Online]. Available: http://www.solaris11.com/?p=440. [Accessed: 04-Jun-2013].
52. Prasad Mujumdar, "Apache Hive, Developers Guide," Mar-2013. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/Home. [Accessed: 10-Jun-2013].
53. Oracle, "MySQL Manual: Running Multiple MySQL Servers on the Same Machine," *MySQL*. [Online]. Available: http://dev.mysql.com/doc/refman/4.1/en/multiple-servers.html. [Accessed: 06-Jul-2013].
54. Esen Sagynov, "Easy MySQL Database Sharding," at MySQL World Conference & Expo, Apr-2013. [Online]. Available: http://www.percona.com/live/mysql-conference-2013/sites/default/files/slides/Esen%20Sagynov%20-

%20Easy%20MySQL%20Database%20Sharding%20with%20CUBRID%20SHARD%20-%202013%20Percona%20PLMCE.pdf, [Accessed: 07-Jul-2013]
55. Petr Dvorak, "MySQL Sharding Block Series: Does Sharding Make Sense on a Single Machine," *Scalebase*, May-2013. [Online]. Available: http://www.scalebase.com/mysql-sharding-blog-series-does-sharding-make-sense-on-a-single-machine/. [Accessed: 06-Jul-2013].
56. Robert H. Dolin, Liora Alschuler, Sandy Boyer, Calvin Beebe, Fred M. Behlen, Paul V. Biron, and Amnon Shabo, "HL7 Clinical Document Architecture," *Journal of the American Medical Informatics Association*, DOI 10.1197/jamia.M1888, pp. 31–39, Feb-2006. [Online]. Available at: ssr-anapath.googlecode.com/files/CDAr2.pdf. [Accessed: 05-May-2013].
57.Ruslan Mukhammadov, *"Fake" health data genetor*. Source code. Jun-2013. [Online]. Available at: https://github.com/ruslanm/data-generator. [Access: 08-Jul-2013].
58. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, 'Benchmarking cloud serving systems with YCSB', in Proceedings of the 1st ACM symposium on Cloud computing, New York, NY, USA, 2010, pp. 143–154, DOI:10.1145/1807128.1807152, [Online]. Available at: http://doi.acm.org/10.1145/1807128.1807152. [Accessed: 01-Jul-2013].
59. Brian F. Cooper, "YCSB," Dec-2012. [Online]. Available: https://github.com/brianfrankcooper/YCSB. [Accessed: 01-Jul-2013].
60. Apache Foundation, "Apache HBase Performance Tuning," Apr-2013. [Online]. Available: http://hbase.apache.org/book/performance.html. [Accessed: 07-Jul-2013].
61. John D.C. Little and Stephen C. Graves, "Little's Law," Massachusetts Institute ofTechnology, Building Intuition: Insights From Basic_Operations Management Models and Principles, 2008, pp. 81–100, DOI: 10.1007/978-0-387 -73699-0, [Online]. Available at: http://web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf. [Accessed: Jul-13-2013].
62. Amdahl Gene, "Validity of the single processor approach to achieving large scale computing capabilities," in Proceedings of AFIPS Conference, IBM Sunnyvale_, California, 1967, pp. 483–485. [Online]. Available at: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf [Accessed: Jul-13-2013].

Appendix A

Sample from Generated Test Data

Physical Examination (Vital signs)	
User Identifier	63481968342523200+62371
Timestamp (in milliseconds)	1378375823000
Pulse Rate	101
SpO2	96
Body Temperature	36.3
Blood Pressure	120
Respiration Rate	15
Blood Glucose	91
Vital Capacity	4.5
Forced Expiratory Flow	39
Forced Inspiratory Flow	42
Tidal Volume	488
End-tidal CO2	5.3
Gait Speed	1.1

Appendix B

RDBMS and Scalable RPMS Prototype Measurements and Chart Representation of each Measurement

B.1 MySQL and HBase Benchmark Statistics

Table B.1-1: MySQL Benchmark with 10 million Rows and 1 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	2,008,661.0	1,605,094.0	1,717,842.0	7,264,402.0
Rows (Millions)	10	10	10	10
Operations (Millions)	1	1	1	1
Thread Count	20	20	20	20
Throughput (ops/sec)	497.844	623.017	582.126	137.658
Average Latency (ms) (read)	6.023	54.492	57.857	299.535
(write)	25.611	NA	0.042	0.101
Minimum Latency (ms) (read)	0	0.009	0.084	0.473
(write)	0	NA	0.005	0.018
Maximum Latency (ms) (read)	3,961.341	999.235	1,027.938	1,878.731
(write)	6,055.092	NA	1.533	501.883

Table B.1-2: MySQL Benchmark with 50 million Rows and 10 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	129,342,237.0	106,463,934.0	114,735,462.0	901,463,235.0
Rows (Millions)	50	50	50	50
Operations (Millions)	10	10	10	10
Thread Count	100	100	100	100
Throughput (ops/sec)	77.314	93.928	87.157	11.093
Average Latency (ms) (read)	107.593	543.088	593.003	2,209.495
(write)	156.239	NA	1.116	1.901
Minimum Latency (ms) (read)	0.937	0.722	0.587	1.769
(write)	1.247	NA	0.222	0.336
Maximum Latency (ms) (read)	21,447.352	4,052.356	7,096.456	19,346.228
(write)	34,534.082	NA	44.623	2,367.2

Table B.1-3: HBase Benchmark on a Cluster of Single Machine with 10 million Rows and 1 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	2,681,339.0	4,923,532.0	3,709,891.0	19,163,908.0
Rows (Millions)	10	10	10	10
Operations (Millions)	1	1	1	1
Thread Count	20	20	20	20
Throughput (ops/sec)	372.95	203.106	269.55	52.181
Average Latency (ms) (read)	106.737	98.268	77.857	402.828
(write)	0.205	NA	0.053	0.217
Minimum Latency (ms) (read)	0.232	0.322	0.141	0.792
(write)	0	NA	0.007	0.039
Maximum Latency (ms) (read)	10,943.329	1,164.307	1,039.029	2,867.829
(write)	5,758.592	NA	2.814	629.926

Table B.1-4: HBase Benchmark on a Cluster of Single Machine with 50 million Rows and 10 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	135,731,469.0	257,662,092.0	188,018,663.0	977,302,292.0
Rows (Millions)	50	50	50	50
Operations (Millions)	10	10	10	10
Thread Count	100	100	100	100
Throughput (ops/sec)	73.675	38.81	53.186	10.232
Average Latency (ms) (read)	1,078.004	761.944	780.906	2,968.392
(write)	0.312	NA	1.442	1.618
Minimum Latency (ms) (read)	0.639	0.893	0.796	2.266
(write)	0.141	NA	0.203	0.454
Maximum Latency (ms) (read)	41,226.053	8,046.395	9,194.734	20,932.729
(write)	17,432.894	NA	12.321	3,021.49

Table B.1-5: HBase Data Load on a Cluster of Two Machines

Million rows	1	10	50
Elapsed Time (ms)	56,022.0	866,528.0	77,623,955.0
Thread Count	10	20	100
Throughput (ops/sec)	17,850.13	11,540.308	644.131
Average Latency (ms)	0.566	2.009	5.993
Minimum Latency (ms)	0	0.003	0.11
Maximum Latency (ms)	8,412.551	30,120.693	46,523.098

Table B.1-6: HBase Benchmark on a Cluster of Two Machines with 1 million Rows and 500,000 Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	37,443.0	72,233.0	56,383.0	232,717.0
Rows (Millions)	1	1	1	1
Operations (Millions)	0.5	0.5	0.5	0.5
Thread Count	10	10	10	10
Throughput (ops/sec)	13,353.631	6,922.044	8,867.921	2,148.532
Average Latency (ms) (read)	1.103	1.934	1.119	5.275
(write)	0.124	NA	0.011	0.03
Minimum Latency (ms) (read)	0.81	0.65	0.72	0.199
(write)	0	NA	0.004	0.004
Maximum Latency (ms) (read)	1,834.639	1,783.246	2,204.003	795.99
(write)	2,993.235	NA	1.8	7.118

Table B.1-7: HBase Benchmark on a Cluster of Two Machines with 10 million Rows and 1 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	1,545,854.0	2,670,352.0	2,130,270.0	10,920,622.0
Rows (Millions)	10	10	10	10
Operations (Millions)	1	1	1	1
Thread Count	20	20	20	20
Throughput (ops/sec)	646.892	374.482	469.424	91.57
Average Latency (ms) (read)	56.043	53.235	40.23	222.435
(write)	0.11	NA	0.03	0.125
Minimum Latency (ms) (read)	0.102	0.14	0.05	0.693
(write)	0	NA	0	0.019
Maximum Latency (ms) (read)	6,239.231	784.239	701.424	1,782.093
(write)	3,664.291	NA	1.703	314.992

Table B.1-8: HBase Benchmark on a Cluster of Two Machines with 50 million Rows and 10 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	75,600,725.0	139,688,149.0	107,045,603.0	563,842,224.0
Rows (Millions)	50	50	50	50
Operations (Millions)	10	10	10	10
Thread Count	100	100	100	100
Throughput (ops/sec)	132.274	71.588	93.418	17.736
Average Latency (ms) (read)	555.348	429.537	399.036	2,012.436
(write)	0.138	NA	1.001	0.931
Minimum Latency (ms) (read)	0.382	0.52	0.448	1.722
(write)	0.08	NA	0.11	0.311
Maximum Latency (ms) (read)	25,775.359	5,192.825	5,002.372	12,523.109
(write)	9,443.892	NA	7.392	1,888.327

Table B.1-9: HBase Data Load on a Cluster of Three Machines

Million rows	1	10	50
Elapsed Time (ms)	38,529.0	584,544.0	52,515,030.0
Thread Count	10	20	100
Throughput (ops/sec)	25,954.476	17,107.352	952.108
Average Latency (ms)	0.401	1.692	3.66
Minimum Latency (ms)	0	0	0.082
Maximum Latency (ms)	6,082.442	21,412.523	68,252.664

Table B.1-10: HBase Benchmark on a Cluster of Three Machines with 1 million Rows and 500,000 Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	25,930.0	48,848.0	38,878.0	161,851.0
Rows (Millions)	1	1	1	1
Operations (Millions)	0.5	0.5	0.5	0.5
Thread Count	10	10	10	10
Throughput (ops/sec)	19,282.684	10,235.834	12,860.744	3,089.261
Average Latency (ms) (read)	0.585	1.303	0.92	3.483
(write)	0.081	NA	0.007	0.011
Minimum Latency (ms) (read)	0.731	0.461	0.494	0.133
(write)	0	NA	0.001	0.001
Maximum Latency (ms) (read)	1,244.823	1,125.883	1,604.352	532.798
(write)	2,000.252	NA	1.203	5.339

Table B.1-11: HBase Benchmark on a Cluster of Three Machines with 10 million Rows and 1 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	1,065,939.0	1,845,835.0	1,468,923.0	7,584,197.0
Rows (Millions)	10	10	10	10
Operations (Millions)	1	1	1	1
Thread Count	20	20	20	20
Throughput (ops/sec)	938.14	541.76	680.771	131.853
Average Latency (ms) (read)	34.331	37.505	28.833	155.006
(write)	0.076	NA	0.009	0.08
Minimum Latency (ms) (read)	0.699	0.892	0.028	0.466
(write)	0	NA	0	0.014
Maximum Latency (ms) (read)	4,368.384	556.844	599.34	1,380.627
(write)	2,934.227	NA	1.352	227.534

Table B.1-12: HBase Benchmark on a Cluster of Three Machines with 50 million Rows and 10 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	52,161,239.0	96,420,325.0	73,354,435.0	394,456,929.0
Rows (Millions)	50	50	50	50
Operations (Millions)	10	10	10	10
Thread Count	100	100	100	100
Throughput (ops/sec)	191.713	103.713	136.324	25.351
Average Latency (ms) (read)	392.593	285.522	266.747	1,402.808
(write)	0.09	NA	0.731	0.629
Minimum Latency (ms) (read)	0.264	0.222	0.297	1.221
(write)	0.051	NA	0.767	0.184
Maximum Latency (ms) (read)	16,648.749	3,983.002	3,034.992	7,666.521
(write)	7,500.466	NA	4.92	1,342.662

Table B.1-13: HBase Data Load on a Cluster of Four Machines

Million rows	1	10	50
Elapsed Time (ms)	29,221.0	443,567.0	39,718,284.0
Thread Count	10	20	100
Throughput (ops/sec)	34,221.964	22,544.508	1,258.866
Average Latency (ms)	0.298	1.209	2.712
Minimum Latency (ms)	0	0	0.059
Maximum Latency (ms)	4,427.629	15,821.935	50,121.821

Table B.1-14: HBase Benchmark on a Cluster of Four Machines with 1 million Rows and 500,000 Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	19,628.0	36,077.0	28,942.0	124,445.0
Rows (Millions)	1	1	1	1
Operations (Millions)	0.5	0.5	0.5	0.5
Thread Count	10	10	10	10
Throughput (ops/sec)	25,473.813	13,859.246	17,275.931	4,017.839
Average Latency (ms) (read)	0.188	1.091	0.77	2.523
(write)	0.053	NA	0.002	0.007
Minimum Latency (ms) (read)	0.598	0.351	0.494	0.133
(write)	0	NA	0	0
Maximum Latency (ms) (read)	0,844.523	813.939	1,336.72	387.552
(write)	1,681.628	NA	0.921	2.917

Table B.1-15: HBase Benchmark on a Cluster of Four Machines with 10 million Rows and 1 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	819,219.0	1,426,036.0	1,139,722.0	6,035,827.0
Rows (Millions)	10	10	10	10
Operations (Millions)	1	1	1	1
Thread Count	20	20	20	20
Throughput (ops/sec)	1,220.675	701.245	877.407	165.677
Average Latency (ms) (read)	21.933	28.992	21.352	112.519
(write)	0.059	NA	0.006	0.053
Minimum Latency (ms) (read)	0.552	0.667	0.023	0.338
(write)	0	NA	0	0.009
Maximum Latency (ms) (read)	3,209.882	384.205	470.425	992.83
(write)	2,032.552	NA	0.91	142.098

Table B.1-16: HBase Benchmark on a Cluster of Four Machines with 50 million Rows and 10 million Operations.

	Workload A Update Heavy (50/50 read/update)	Workload C Read Only (100% read)	Workload D Read Latest (95/5 read/write)	Workload E Range Scan (95/5 scan/write)
Elapsed Time (ms)	41,026,048.0	76,139,177.0	57,634,569.0	325,590,021.0
Rows (Millions)	50	50	50	50
Operations (Millions)	10	10	10	10
Thread Count	100	100	100	100
Throughput (ops/sec)	243.748	285.522	173.507	30.714
Average Latency (ms) (read)	291.423	221.258	205.811	991.004
(write)	0.067	NA	0.522	0.404
Minimum Latency (ms) (read)	0.193	0.18	0.2	0.822
(write)	0.039	NA	0.519	0.135
Maximum Latency (ms) (read)	13,623.56	2,422.524	2,002.516	5,622.092
(write)	5,622.552	NA	3.042	1,001.005

B.2 Chart Representation of Benchmark Statistics

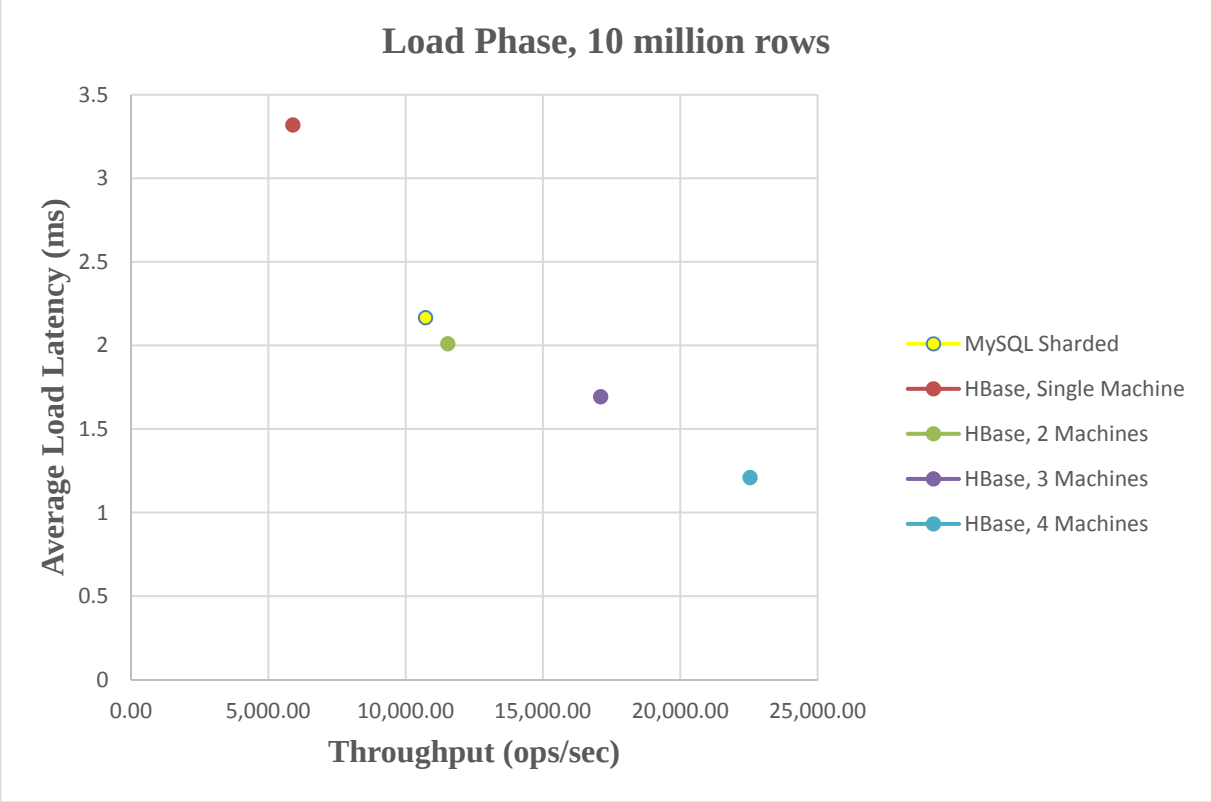


Figure B.2-1: 10 GB of Data Load Performance Differences

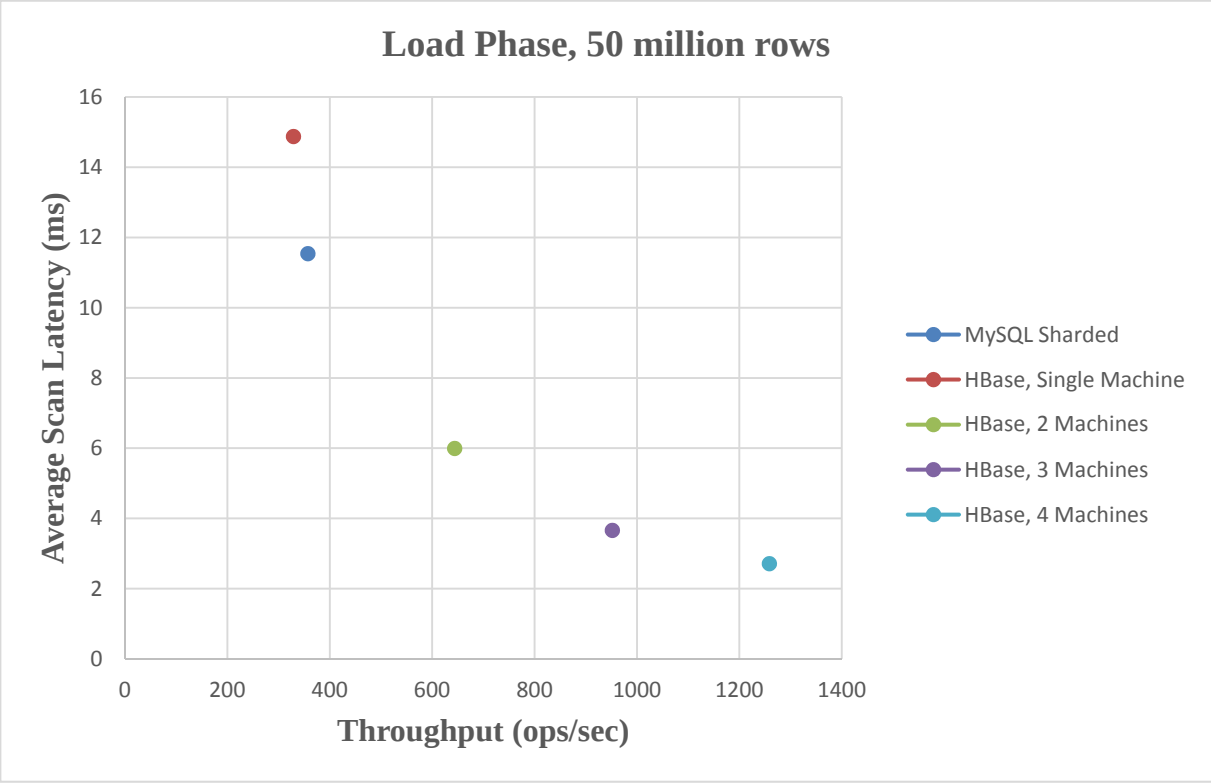


Figure B.2-2: 50 GB of Data Load Performance Differences

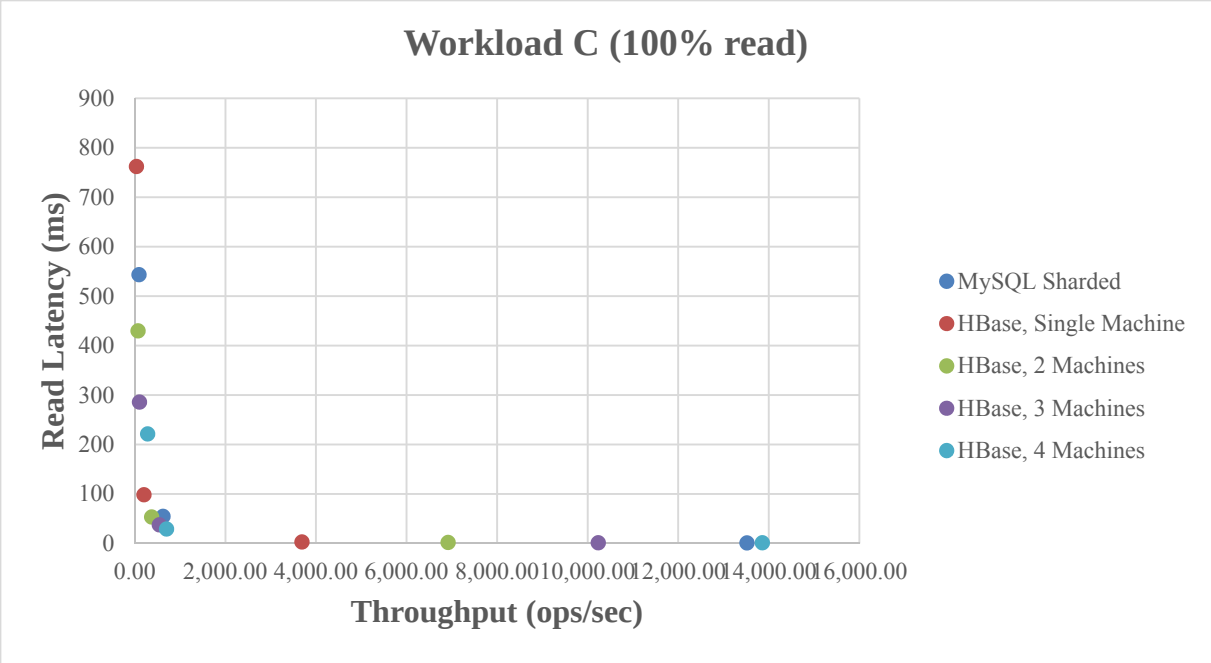


Figure B.2-3: WorkloadC Read Latency vs. Throughput Benchmark Statistics

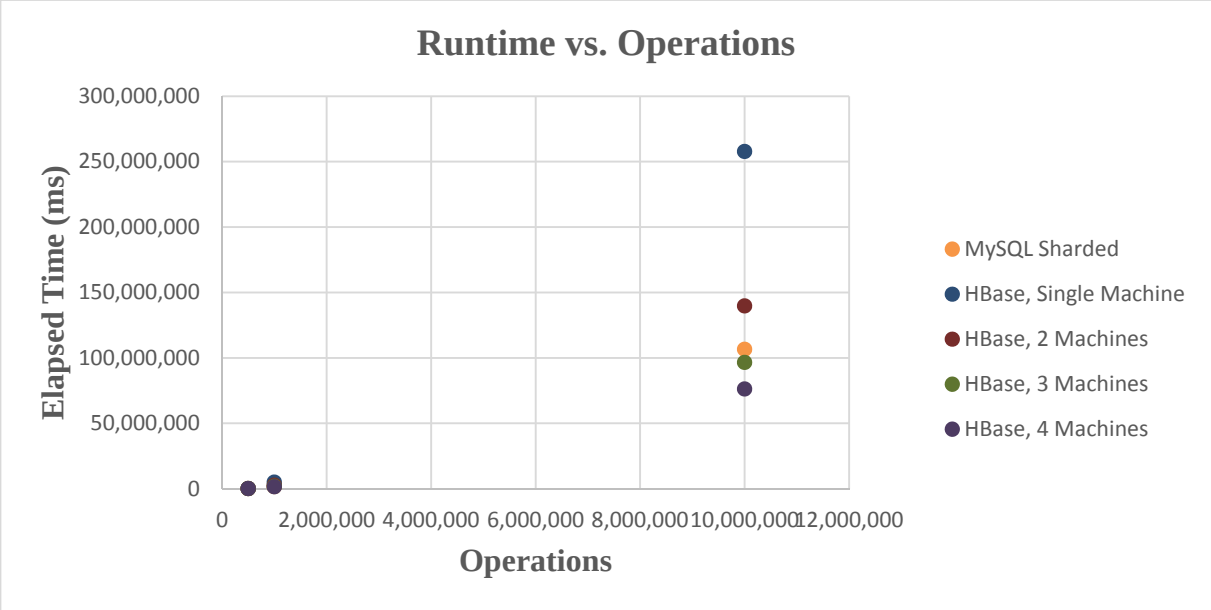


Figure B.2-4: Elapsed Time vs. Number of Operations Chart on WorkloadC

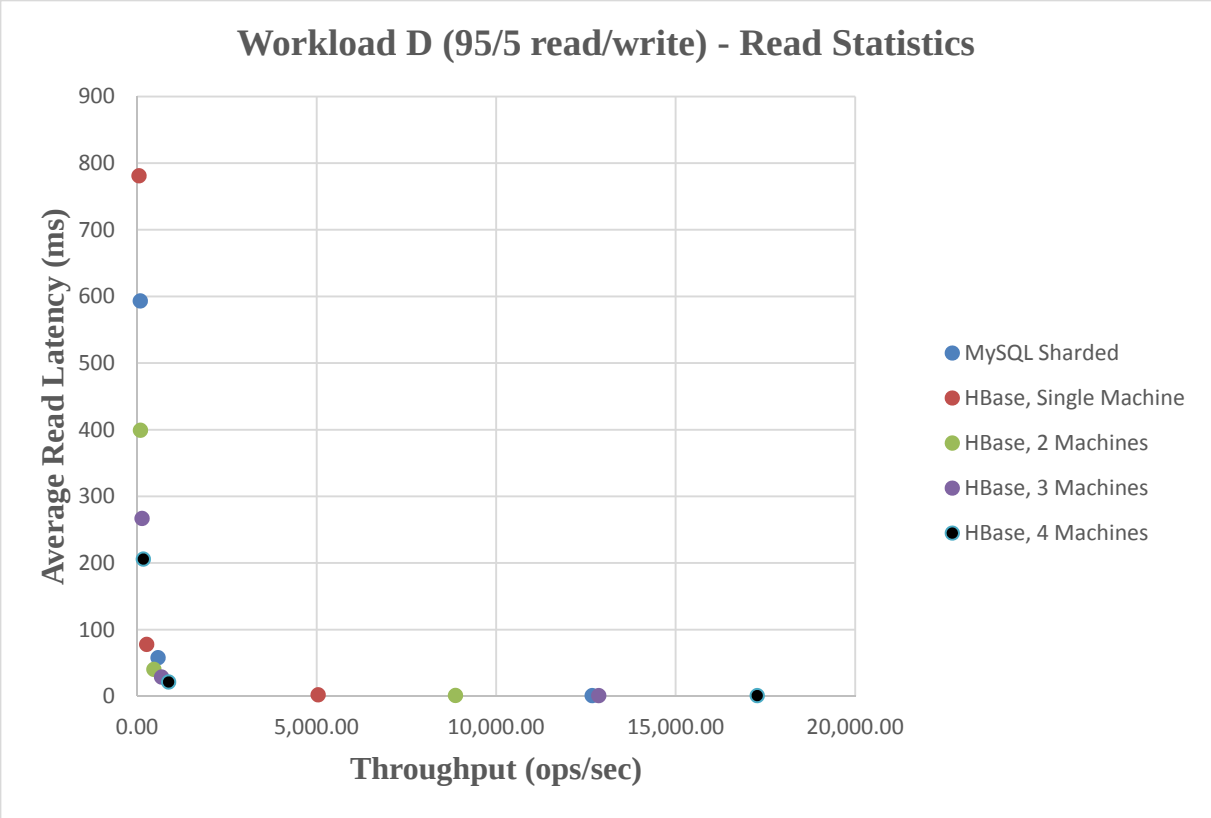


Figure B.2-5: WorkloadD Read Latency vs. Throughput Benchmark Statistics

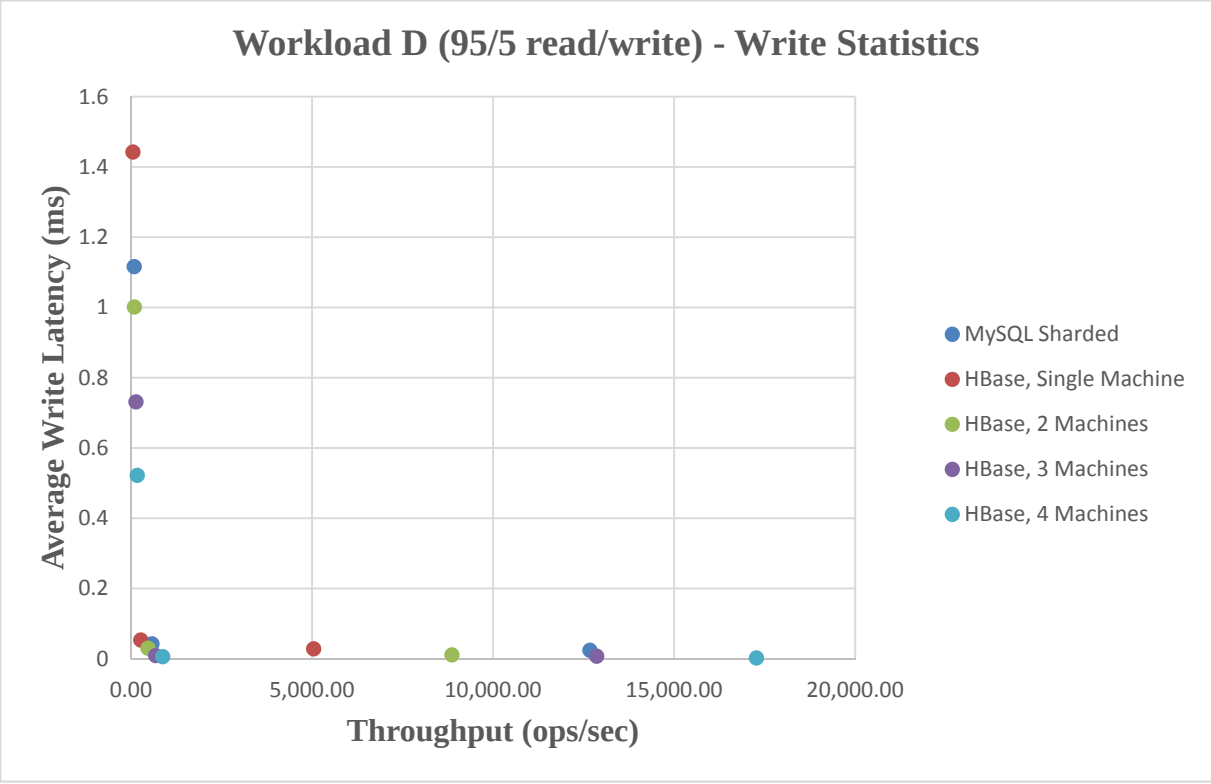


Figure B.2-6: WorkloadD Write Latency vs. Throughput Benchmark Statistics

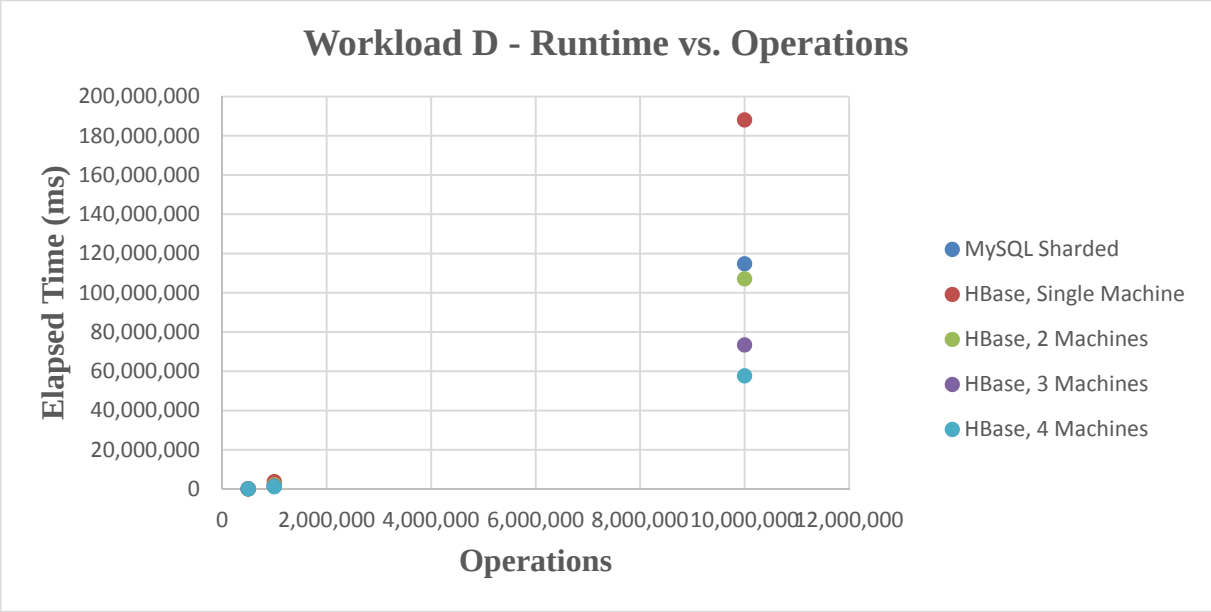


Figure B.2-7: Elapsed Time versus the Number of Operations Chart on WorkloadD

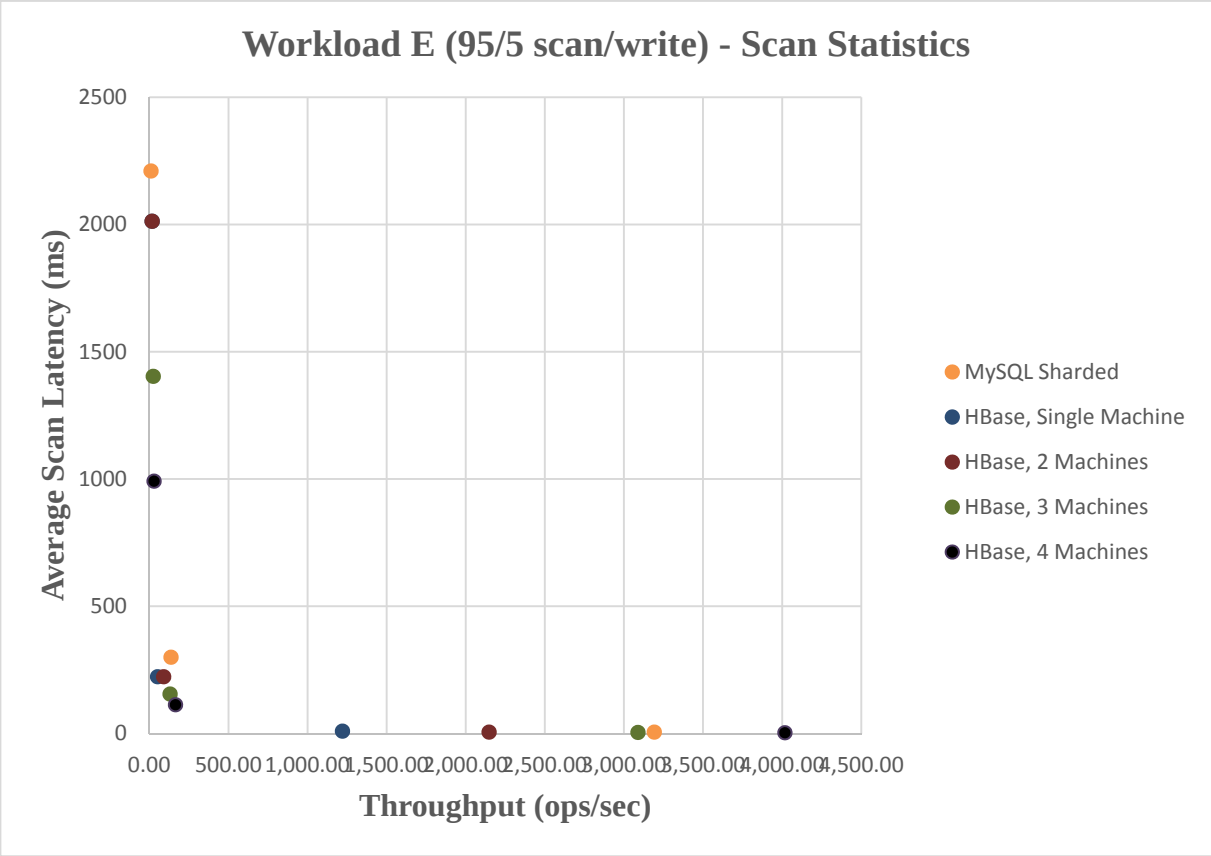
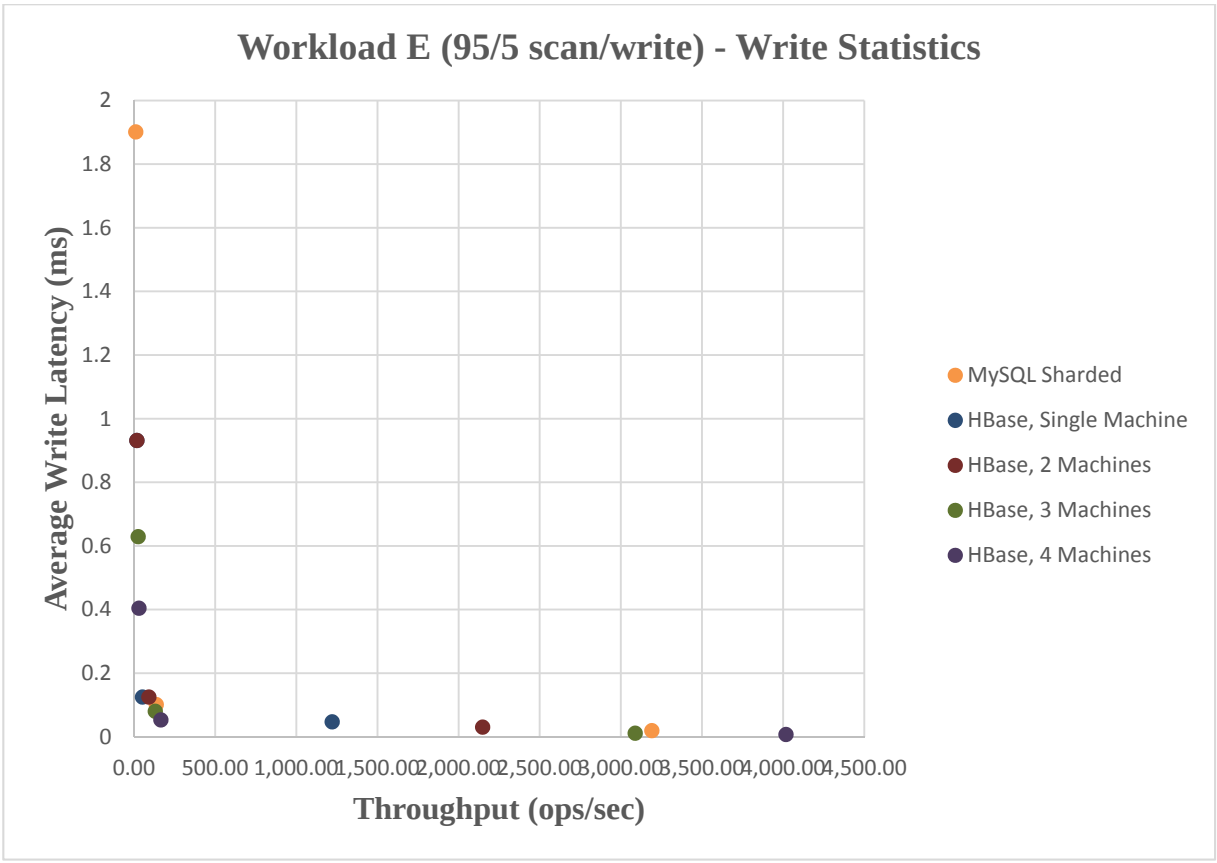


Figure B.2-8: WorkloadE Scan Latency vs. Throughput Benchmark Statistics



FigureB.2-9: WorkloadE Write Latency vs. Throughput Benchmark Statistics

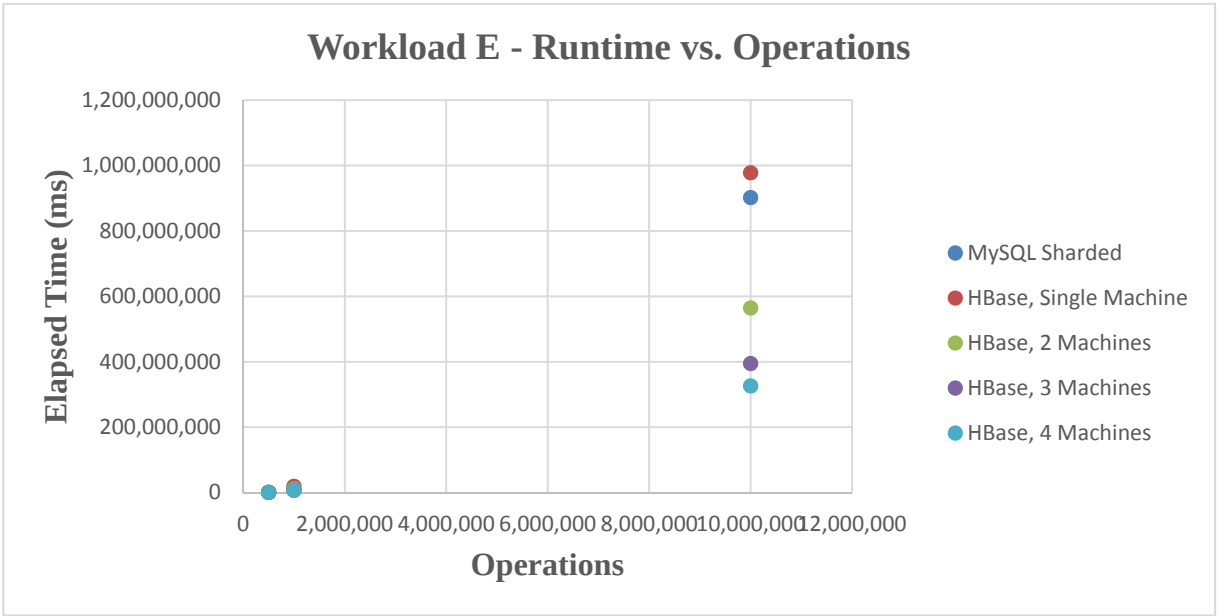


Figure B.2-10: Elapsed Time versus the Number of Operations Chart on WorkloadE

TRITA-ICT-EX-2013:195

www.kth.se