An Empirical Evaluation of Big Data Technologies For Federated Sensor Services

by

Siva Venkat Gogineni

(Under the direction of Lakshmish Ramaswamy)

Abstract

Sensors utilization increased significantly in the past few years from a limited number to around 50 billion, as a result there is a multitudinal increase in data generation that is driving the boom of big data technologies. Federated sensor framework has the potential to utilize the existing big data technologies to efficiently process and store enormous amounts of data. Since, the field of big data is relatively new there are many technologies that are still unexplored and unknown. The focus of this study is to compare several big data technologies and tools. The study provides information about the tools functionalities and compares the results of several queries that are designed to test their performance. At the end of study we provide a set of technologies that we think are a good match for the federated sensor framework.

Index words: Federated Sensor services, SDQML, Big Data, NoSQL, SPARK, SML, Hadoop

An Empirical Evaluation of Big Data Technologies For Federated Sensor Services

by

Siva Venkat Gogineni

B.E., Anna University, 2008

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2014

©2014

Siva Venkat Gogineni

All Rights Reserved

An Empirical Evaluation of Big Data Technologies For Federated Sensor Services

by

Siva Venkat Gogineni

Approved:

Major Professor: Lakshmish Ramaswamy

Committee: John A. Miller
Deepak Mishra

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2014

Acknowledgments

I am greatly thankful to my major professor, Dr. Ramaswamy for his continuous support and guidance during my study at UGA. He is a wonderful professor and always encourages his students. I would also like to thank Dr. Mishra for his valuable suggestions and support. I am thankful to Dr. Miller, who has introduced me to Big Data and helped in understanding the concepts in my third semester. Finally, I would like to thank my colleagues who have helped me on this project: Vinay Kumar Boddula, Victor lawson, Satya Vikas, Dileep Bodanki and Gowtham Penematsa.

Contents

Acknowledgments **iv**

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

2 Motivation And Background **4**

2.1 Big Data 6

2.1.1 Computation model 6

2.1.2 Storage Model 8

3 Architectural Overview **11**

3.1 SDQ-ML 11

3.2 Federated Sensor Framework 12

4 Federated Sensor Framework Architecture **14**

4.1 Ad hoc query processing system 20

4.2 Continuous query processing system 20

5 Experimental Results **22**

5.1 Experimental Setup 22
5.2 Experimental Dataset 23
5.3 Experiments on data stores: 23
5.4 Experiments on machine to machine interaction mechanisms: 31
5.5 Experiments on Realtime Processing system: 31

6 Related work **35**

7 Conclusion **37**

Bibliography **37**

Appendices **42**

Appendix A SAMPLE SDQML **42**

Appendix B Experiments Data **46**

List of Figures

2.1 CAP theorem 9

3.1 Federated Sensor Framework 13

4.1 Federated Sensor Framework 15

4.2 Flow chart of Federated Sensor Framework 16

5.1 Random Read operation on various data stores on VM 24

5.2 single random read operation on various data stores on RaspberryPi 25

5.3 Random Read operations on various data stores on VM 25

5.4 Random Read operations on various data stores on RaspberryPi 26

5.5 Bulk Write operations on various data stores on RaspberryPi 27

5.6 Write operation on various data stores on VM 28

5.7 Write operation on various data stores on RaspberryPi 28

5.8 Bulk Read operations on various data stores 29

5.9 Bulk Read operations on various data stores 30

5.10 Thrift vs Web services on VM 32

5.11 Thrift vs Web services on RaspberryPi 32

5.12 Logistic Regression on Spark and Hadoop 33

5.13 Word Count on Spark and Storm 33

5.14 Grep on Spark and Storm 34

List of Tables

3.1 SFDL 12

3.2 SURL 12

4.1 Features of BaseX and Sedna 18

B.1 Random Read: Query 1 record from MongoDb 46

B.2 Random Read: Query 100 records from MongoDb 47

Chapter 1

Introduction

The Central Nervous System for the Earth (CeNSE) [37], which is a wireless sensing system developed by HP and Shell is estimated to reach 1 trillion sensors that can generate data to the magnitude of 50 zettabytes(10^{21} bytes) per year by 2020. The Wireless World Research organization [36] is predicted to expand about 7 trillion sensors by 2017. ON Worlds research has predicted that it would ship 18.2 million health and wellness wireless sensors in 2017. Wireless Sensor Network [33] Markets is predicted to reach $12 billion Worldwide by 2020 from $2.7 billion in 2013. IBM forecasts that sensors for context-aware computing will grow to a trillion sensors by 2015. Cisco estimates that IOT [46] will drive a $14 trillion business worldwide by 2020. Overall, it can be concluded from the above projected growth estimates that there will be an exponential growth of sensor devices in the coming years that can potentially create multitrillion dollar business worldwide.

If these business and technology trends are to be followed, then the next few years will very likely see a cloud-based servitization of domain sensing functionalities. Service providers could independently install and maintain sensors across the globe, and the data from these sensors would be exposed as a service to data consumers or to domain specific applications.

Consumers can also embed these data feeds into their applications. It is noteworthy that this is a manifestation of the Data as a Service (DaaS) model, the viability of which has been validated by the recent emergence of several data markets including Azure DataMarket) [32] and InfoChimps) [34]. However, currently most data markets only host static and precollected (mostly relational) data.

We envision a federated sensor services paradigm which demands many advanced capabilities that are lacking in the current state of the art research. First and foremost, sensor service platforms will necessarily have multiple distinct stakeholders. There will be multiple sensor service providers possibly with very different capabilities and multiple service consumers with distinct sensor feed requirements. For example, each domain application (sensor feed consumer) should easily be able to locate and utilize sensor feeds that best fit its data needs. Simultaneously, the sensor service providers must be able to collaborate as well as compete. Federation and service-orientation become very important in this regard. Second, domain applications generally have specific data quality (DQ) requirements. As mentioned above sensor services are essentially data services. Hence, service quality, to a large extent is determined by the data quality of the sensor feeds. Unfortunately, the notion of DQ is extremely weak in most current wireless sensor network (WSN) [33] and sensor web frameworks. Third, sensor service platforms should be highly scalable; the number of feeds that need to be handled could easily be in the thousands, and they may originate from anywhere in the globe. Some may even originate from remote regions such as volcanic mountains. This means that the framework must be decentralized as well as geographically distributed. While the cloud seems to be a natural choice for hosting sensor service platforms, there is surprisingly little work in this direction. To the best of our knowledge, no one has investigated mechanisms to apply data quality toward clouds for sensor service platforms comprehensively.

In this thesis, we outline our vision for a Data Quality-centric, framework for federated sensor services using big data. We explore and benchmark various big data technologies that are required to build such framework for federated sensor services. Furthermore, we provide a list of big data technologies which we think are a better fit for federated sensor framework.

We organize this thesis as follows. In chapter 2, we first motivate the need for a federated sensor service by discussing a few examples from various domains and then we give a background of our work. In chapter 3, we give an overview of the architecture of our work. In chapter 4, we discuss the various existing systems that suit our needs and requirements. In chapter 5, we discuss how the queries are answered by our framework. The performance and benchmark results are given in chapter 6. In chapter 7, we discuss the related work. In Chapter 8, we conclude our work.

Chapter 2

Motivation And Background

In this section, we will first motivate the need for federated sensor services using examples from various domains. We will then provide the necessary background to understand the concepts of big data and technologies associated with it.

The current scenario in the sensor service network requires that consumer install and operate their own private sensor infrastructures. Obviously, this is a very expensive solution. However the federated sensor service paradigm has several clear advantages. First, servitization distributes the installation and operation overheads over all the consumers for a particular sensor feed, and hence reduces the overheads for individual consumers. Second, it enables the businesses and researchers shared access to sensor data in real-time on a truly global scale thus making the sensor data more reliable. Third, it reduces the cost of setting up the infrastructure for businesses specializing in providing high quality sensor data.

However, the sensor service paradigm also poses some significant challenges. As an example, let us consider wind speed sensor feeds. Applications in different domains use the data from wind speed sensors to run various analytics. Weather monitoring applications use these feeds for issuing emergency storm alarms. Wind energy companies rely on these feeds for

planning installations and operations. Atmospheric scientists need them for effects of climate change on wind patterns and many others. Currently, each of these consumers must install and operate its own sensor infrastructure, which results in a lot of resource duplication and unnecessary wastage. It would be ideal to come up with a collaborative strategy for all the organizations to use feeds from the same sensor infrastructure. However, each of these applications have very different requirements with respect to the content and quality of the feed. A storm monitoring application will likely need a real-time sensor feed with minimal delay. Whereas, in case of an offline scientific data analysis application the latency requirement can be relaxed, but may have stringent accuracy and completeness requirements. Moreover, at any given geographical location, there may be different wind-speed sensors that produce the same feed but the accuracy and the frequency of the feed vary based on differences with hardware, quality of the wireless networks and the power source used. For instance, there are a number of different anemometers like the cup anemometer, windmill anemometer, hot-wire anemometer, laser doppler anemometer, etc. All of which may be used to measure the wind speed but each of these classes has distinct DQ (Data Quality) characteristics. Sonic anemometers provide fine temporal resolutions, whereas windmill anemometers provide high accuracy, and hot-wire anemometers yield fine spatial resolution. A federated sensor service should ideally enable the consumers to obtain data that best suits their requirements (it could be both content and quality). In order to achieve this, the framework should satisfy three main design principles:

Simplicity: The framework should be based on a minimal set of powerful abstractions which could be adopted easily by end users and data service providers. User request and server response should be specified in a declarative way using convenient handles. For example, using XML like language as the syntactic framework. All interactions between the stakeholders, framework and end users should take place using this language.

Scalability: A framework that can handle growing amounts of data is said to be scalable. A system or a framework can scale up (vertical) or scale out (horizontal). Scaling up is adding more resources to a single node. Scaling out is adding more nodes to the system. The framework should be designed in such a way that, as the data grows the system should be in a position to scale out efficiently and gracefully.

Lightweight: A system or a protocol that is simpler, faster or has fewer components than its counterparts is said to be lightweight. The framework needs to be lightweight because its stakeholders could range anywhere from simple embedded device to very powerful server. So, regardless of stakeholder, the framework should be designed to work smoothly.

2.1 Big Data

Big data can be defined as the data that exceeds the processing capacity of conventional database systems and computational systems. More often than not big data is associated with the 3V's(Volume, variety, velocity). The volume of the data is so huge that the conventional databases and computation systems cannot process them. The data could either be structured or unstructured data. The rate at which data flows into an system is called velocity. All the big data technologies and tools fall under one of the below categories: 1. Computational Model. 2. Storage Model.

2.1.1 Computation model

In order to process such large volume of data, we need a distributed, fault tolerant and efficient computation system. Hadoop [44], Storm [30] and Spark [31] are few of the most popular products present in the market. Each of these computation models work differently from others. Let us discuss each of them in detail. Hadoop is an open source framework for

large scale data processing. It is designed to solve problems that involves analyzing large data i.e, Petabytes (10^{15} bytes). It is basically a programming model based on Googles Map Reduce [2] paper. It handles large data throughput and supports data-intensive distributed applications. It runs on a collection of commodity servers unlike other computational models. Hadoop has two main components, one for processing large volume of data known as MapReduce and the other being the file system itself to store such large volumes of data known as Hadooop Distributed File System(HDFS). MapReduce is the most trivial component of Hadoop ecosystem. It is a software framework that allows developers to write programs that process massive amounts of data in parallel across a distributed cluster of processors or stand-alone computers. The framework has two main components, mapper and a reducer. Mapper takes a set of data as input and converts it into tuples (key/value pairs). The reducer takes the output from a map as input and combines data tuples of distinct keys into a smaller set of tuples. Hadoop as such cannot be used for realtime stream processing, but Hadoop along with Yarn can be used for realtime stream processing. Pig [22] is an open-source high-level dataflow system. It provides a simple language for queries and data manipulation, Pig Latin is compiled into map-reduce jobs that are run on Hadoop. It is a high level language that abstracts the Hadoop system completely from users. It is a data flow language, not procedural language. It provides common operations like join, group, sort and etc. Hive [21] is another high level language that abstracts from the end uers. It is targeted towards users, who are comfortable with SQL. It is similar to SQL and called HiveQL. Hive as well as Pig [22] are used for batch analytics.

"Storm is a distributed, fault-tolerant, and high-performance realtime computation system that provides strong guarantees on the processing of data". Storm also adds reliable real-time data processing capabilities to Apache Hadoop. A Storm cluster is similar to a Hadoop cluster. In Hadoop we run MapReduce jobs and in Storm we run topologies. Jobs and topologies themselves are different, one key difference is that a MapReduce job eventu-

ally finishes, whereas a topology runs forever. Topologies consists of spouts and bolts unlike jobs which consists of mappers and reducers. Spouts are data connectors that are used to integrate Storm with other data source streams. Currently Storm supports Flume, Kafka, Twitter Api and etc as data connectors. Storm bolts perform the function of ETL (Extract transform and load) and data processing. Storm topology manages the stream computation with spouts and bolts.

Spark is a fast and powerful engine for processing data. It runs on Hadoop [44] clusters through Hadoop YARN or Spark's standalone mode, and it can process data in a file system, using HDFS, HBase, Cassandra, Hive, and any Hadoop input format. It is designed to perform both general data processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning. It also provides Resilient Distributed Datasets (RDD's), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. It also allows users to persist these RDD's to disk if needed. "Spark is up to 100x faster than Hadoop MapReduce in memory, or 10x faster than hadoop MapReduce on disk". This is because in Hadoop, the output of mapper and reducer are persisted to disk and reading from disk itself is a bottleneck. Whereas, Spark overcomes this problem by using RDD's, which allows you to transparently store data in memory and persist it to disk if it's needed.

2.1.2 Storage Model

Why do we need a new storage model when we have been already using RDBMS for quiet some time? When applications already depend on a traditional RBDMS model for data management, it may be sufficient to scale the RDBMS. However, many applications that expect to take advantage of a high-performance, distributed data environment are not suited to consume data from traditional RDBMS. That means we must consider alternate methods for data management and that is NoSQL databases.

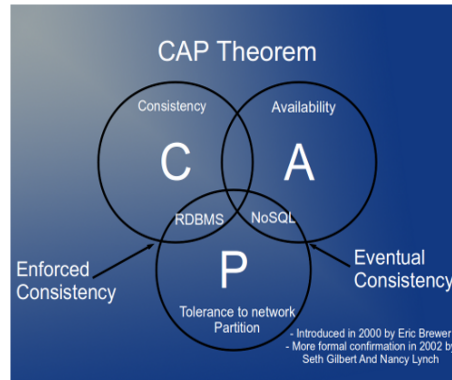

Figure 2.1: CAP theorem

NoSQL databaseS, also called as Not only SQL, provide effective techniques for data management and database design that are useful for very large sets of structured and unstructured data. NoSQL contains a wide range of technologies, algorithms and architectures to solve the issues that relational databases are facing with big data. NoSQL is very useful when an organization needs to analyze and store very large amounts of unstructured data.

NoSQL databases do not follow a relational structure because they intend to allow different models to be adapted to specific types of problems. Although the relaxed approach to modeling and management gives a huge performance improvements for applications, it does not enforce any data constraints unlike traditional databases. As a result there might be inconsistency in data. According to the Brewer's CAP theorem [12], "It is not possible for a distributed system to achieve the properties of consistency, availability, and tolerance to network partitions". NoSQL databases could be broadly classified as:

Key-value stores are the simplest NoSQL databases. Every single item in the database are stored as an Key-value pair. All the keys are unique, i.e, no two keys can be same. Key-value stores include Redis, Dynamo, Membase, etc.

Document stores are basically key/value stores with one major difference. Instead of just storing any blob as value, a document store requires the data to be in a known a format

that the database can understand. Some of the most popular formats are XML, JSON and BSON (Binary JSON). Document stores include CouchDb, MongoDb, BaseX, etc.

Graph databases store data as a graph structure based on graph theory. They contain nodes, edges and relationship between nodes. Graph stores include Neo4J, FlockDb, HyperGraphDB, AllegroGraph, etc.

Columnar database stores content by columns rather than by row and are optimized for queries over large datasets. One of the major benefits of a columnar database is that it helps in compressing the data greatly which makes the aggregation operations like avg, sum, count, min and max extremely fast. Columnar databases are primarily used in data warehousing domain which deals with large volume of data. Columnar stores include Cassandra, HBase, Hypertable, etc.

Chapter 3

Architectural Overview

In this thesis, we present our vision for data quality (DQ)- centric infrastructure for federated sensor service clouds. We provide an architecture in which DQ is ubiquitous throughout the platform. Our work includes a markup language called SDQ-ML (Sensor Data Quality Model Language) for describing sensor data (Sensor Feed Description Language) as well as for domain applications to express their sensor feed requirements (Sensor User Request Language). We have explored the advantages and limitations of various big data technologies, that we think are a good match for federated sensor framework.

3.1 SDQ-ML

Sensor Data Quality Model Language (SDQ-ML) is a superset of Sensor Model Language (SML). It contains all the tags of SML and also contains few additional tags to suit the framework. All the communications within the framework happens through SDQ-ML. Domain applications express their sensor feed requirements using SURL which is a part of SDQ-ML. The data service providers express their feed metadata to the framework using SFDL which is also a part of SDQ-ML. The framework requests for sensor data from the service providers

using SURL. Service providers respond to the request using SURL. SDQ-ML was designed in such a way that it could be easily extended to the growing needs of the framework.

Table 3.1: SFDL

TAGS	Description (S=SensorML element, DQ=SDQ-ML tag)
Name	(S) Sensors reading types: Temp, Light, CO2, Oxygen
Location	(S) Sensor GPS longitude and latitude
Timeliness	(DQ) Delay between data value recording and report to client
Trustworthy	(DQ) Feed collected and processed by secure agencies

Table 3.2: SURL

TAGS	Description (S=SensorML element, DQ=SDQ-ML tag)
Id	(S) Sensor id
Frequency	(DQ) Rate at which readings are reported in the feed
Aggregate	(DQ) Calc average, count , MIN, MAX, SUM of range of values
Range	(DQ) Numeric range values. By Fahrenheit or Celsius

3.2 Federated Sensor Framework

The Federated Sensor Framework needs to address several properties such as scalability, fault tolerance, availability, robustness and framework specific requirements. In order to build such an framework, first we would require a data store to store the actual sensor data in realtime. Secondly, we would require a data store to store and query SDQ-ML (metadata). Thirdly, we would require a machine to machine interaction mechanism to communicate between different stakeholders and within the framework itself. Fourth, we would require a distributed realtime computation system for processing sensor stream data.

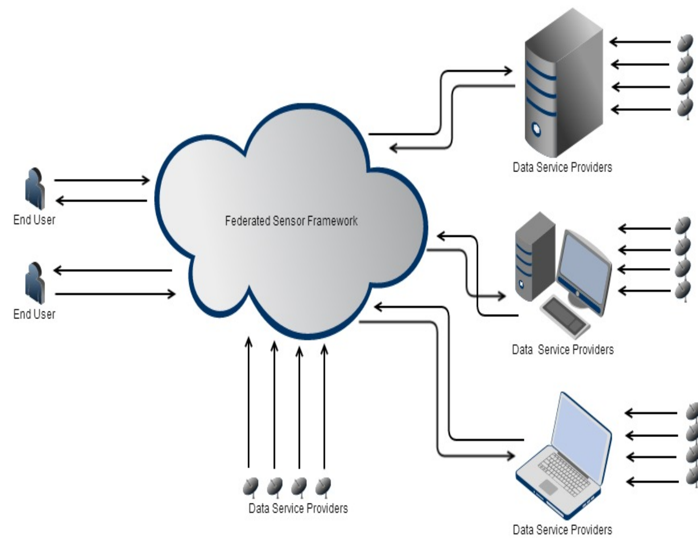

Figure 3.1: Federated Sensor Framework

Chapter 4

Federated Sensor Framework Architecture

Federated sensor framework requires several existing Big data technologies as a part of the framework. Firstly, it requires a data store to store the actual sensor data. Secondly, it requires a data store to store and query SDQ-ML (metadata).Thirdly, it would require a machine to machine interaction mechanism to communicate between different stakeholders and within the framework itself. Fourth, it would require a distributed realtime computation system for processing sensor stream data.

A Data Store is used to store the sensor data from the realtime streams as well as data from service providers. A data store is a repository for persistently storing collections of data, such as a database, a file system or a directory. We have considered a few of the leading data stores such as PostgreSQL [23], MySql [24], CouchDb [25], MongoDb [19] and Hbase [20] which meet our framework requirements. However, PostgreSQL and MySql do not satisfy network partition hence we are left with CouchDb, MongoDb and Hbase. CouchDb may seem like a good choice at first glance. However, CouchDb does not provide complex querying

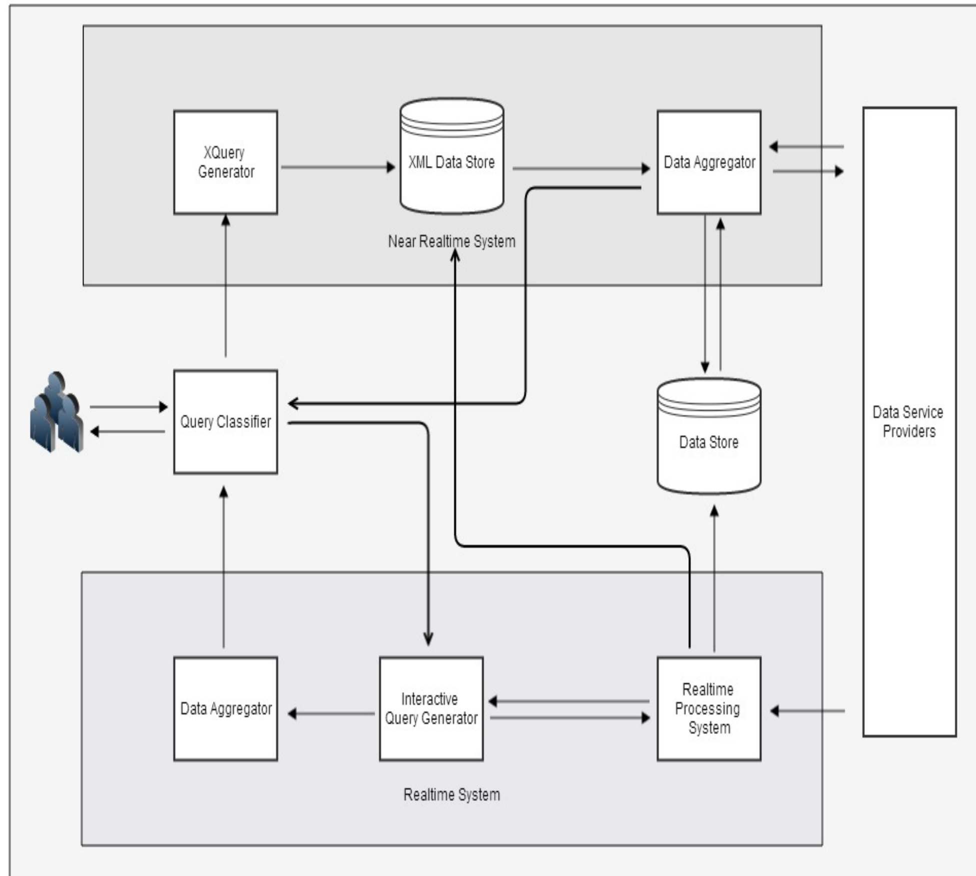

Figure 4.1: Federated Sensor Framework

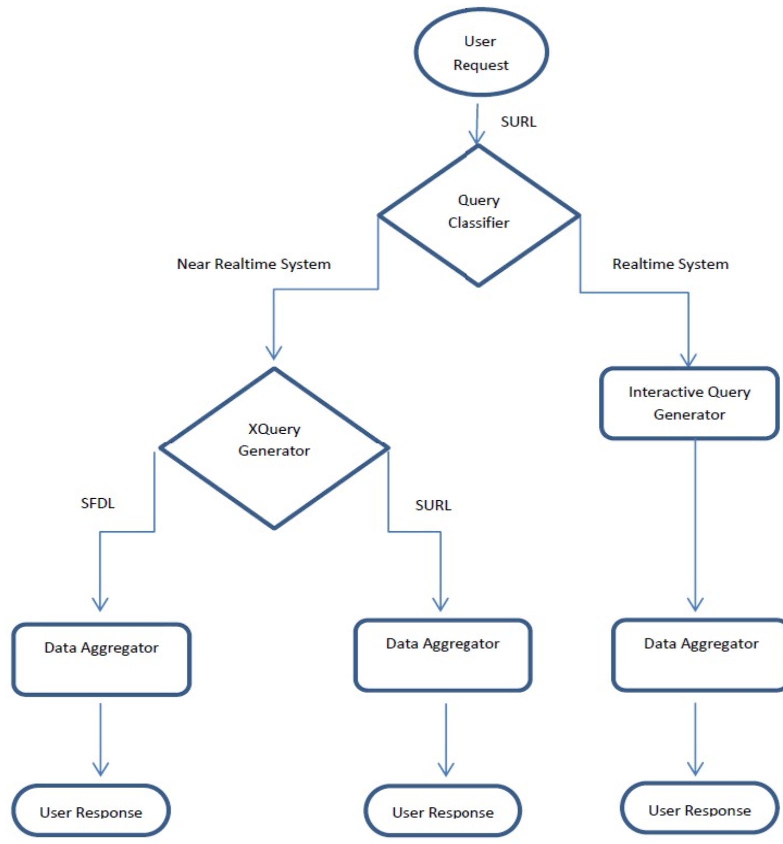

Figure 4.2: Flow chart of Federated Sensor Framework

capabilities as MongoDb and Hbase. CouchDb would be a good option for applications that require static queries by using map reduce. So we are left with two options from the initial set of databases MongoDb and Hbase. Hbase does not support geospatial capabilities unlike MongoDb. So if we need geospatial search capabilities with Hbase then we need to integrate with SOLR [10]. Indexing geospatial data using SOLR on Hbase might look like a feasible solution but by doing so, we will have a performace drop. We think MongoDb is a better choice for our framework, even though MongoDb does not support triggers unlike Hbase.

XML Data store is required to store, maintain and query SDQ-ML (metadata) pushed by data service providers. We query the data store more often than we write to it. There are two types of XML data stores namely XML-enabled database and native XML database (NXD). Most of the traditional databases such as IBM DB2 [40], Microsoft SQL Server [39], Oracle Database [41] and PostgreSQL support XML but the problem with these traditional databases is that they do not support complex querying capabilities for XML as they are not just meant for XML data. Moreover, some of the databases also map XML schema to database schema rather than directly storing as plain XML. On the other hand, NXDs are specifically developed for XML. Hence they are more suitable for our framework than XML-enabled databases. Most NXDs support XPath [42] and XQuery [43] for querying XML documents. XPath is most commonly used query language for NXDs. The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides functionality for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document. However, XPath lacks functionality like grouping, sorting, cross document joins, etc. XQuery has overcome

these shortcomings. According to W3C, XQuery is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath, XQL, XML-QL, SQL and OQL. XQuery is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents. XQuery is to XML like SQL is to database tables. XQJ is a XQuery API for java. BaseX [27] and Sedna [26] are the most popular and widely used databases in the world of native XML databases. Sedna does not support XQuery, XQJ and querying multiple documents. We think BaseX is a better choice for our framework

Table 4.1: Features of BaseX and Sedna

Name	XQuery 3.0	XQuery Update	XQuery Full Text	XPath	XQuery
BaseX	Yes	Yes	Yes	Yes	Yes
Sedna	No	Yes	Yes	No	No

We need a realtime computation system to compute and analyze the sensor stream data coming from several sensors which are deployed across the world. The system should be scalabale, fault tolerant, fast and should guarantee data processing. There are not much of such systems apart from Storm [30] and Spark [31]. Storm is relatively slower when compared to Spark and does not support interactive querying mechanism. We think Spark is a better choice for our framework.

Our framework consists of several individual components interacting and sharing data between each other. Hence, it becomes important in choosing a scalable, efficient and fast mechanism for communication between these components. Web services are the most popular and easier to achieve communication between electronic devices/systems. Web services are easier to develop, thanks to the tremendous interest in them and the support for them in developer tools and through libraries and frameworks. However, if your payload is small,

the overhead you create with Web services is prohibitive. Remote Procedural calls using TCP/IP sockets for communications are a better option, but the problem of using TCP/IP sockets for communications is it is comparatively harder to develop. Hence we choose Apache Thrift for creating custom RPC calls as it is fast, lightweight and easy to implement. Thrift is also used by Hadoop, Hbase, Cassandra and Hypertable.

The Federated Sensor framework is capable of answering continuous and ad hoc queries. It broadly consists of a Continuous query processing system and a Ad hoc query processing system. Ad hoc query processing system is used for answering queries that involves historic data. It constitutes of XQuery generator, XML data Store and Data Aggregator. Whereas, a Continuous query processing system is used to answer queries that involves realtime data. It constitutes of Interactive Query generator, Realtime Processing system and Data Aggregator. Let us discuss the functionality and role of each of these components in detail.

End users or domain applications query the federated sensor framework for sensor data using SURL. SURL as shown in table 3.2 on page 12 provides a wide range of DQ-metrics which can be used by the domain applications and the framework itself to express their feed requirements. Shown below, in listing 4.1, is an example SURL which requests for temperature data in Athens, GA over a period of 30 days with a frequency of 0.0416 per hour.

The Query classifier validates and parses the given query (SURL) using JAXB against the SDQ-ML XSD. It determines how any given query can be answered, i.e, in realtime or in near realtime. It decides this based on the user request. In the example SURL, shown above in listing 4.1, the user requests for historic data and hence the request is processed by near realtime system.

4.1 Ad hoc query processing system

Xquery Generator generates an XQuery from the SURL to query against the XML data store. It queries against the SURL database present in the XML data store, to check if the request was answered earlier. If so then it would answer the query from the data store present within framework itself. Otherwise, it then queries against the SFDL (Metadata) database to get a list of service providers that contain the requested data. Then the list of service providers along with SURL are sent to the Data Aggregator.

The XML data store is used to store the SFDL (metadata) of the service providers for querying and also, stores SURL for caching purposes. The data store is used to persist realtime streaming sensor data as well as data recieved from data service providers while answering queries. It might store several millions or even billion of records.

The Data Aggregator is responsible for querying the data from the service providers or from the internal data store and aggregate the data based on the user request. It also persists the data acquired from various data service providers into the data store for later usage and persists SURL into the XML data store to serve as a cache for the future use. It is also responsible for validating and storing SFDL pushed by data service providers.

4.2 Continuous query processing system

The Interactive Query Generator is responsible for generating a query from the SURL to query against the RDD present in Spark, which is a realtime processing system. Spark is used to process and analyze the streams of data coming into the system. It also persists the data into MongoDb and generates SFDL for future use.

Data Aggregator is responsible for querying the data from the service providers and aggregate the data based on the user request. It also persists the data acquired from various data service providers into the data store for later usage, and persists SURL into the XML data store to serve as a cache for the future use. It is also responsible for validating and storing SFDL pushed by data service providers.

Listing 4.1: SURL

```
1 <?xml version="1.0"?>
2 <SDQML>
3 <!--Sensor User Request Language-->
4 <SURL><sensor>
5 <id>YSI_RG_0001</id>
6 <name>temperature</name>
7 <location>
8 <latitude>33.95</latitude>
9 <longitude>83.38</longitude>
10 </location>
11 <duration start="2012-05-30T09:00:00" end="2012-06-30T09:00:00"/>
12 <!-- No of readings per hour-->
13 <frequency>0.0416</frequency>
14 </sensor></SURL>
15 </SDQML>
```

Chapter 5

Experimental Results

In this chapter, we discuss about the experiments that were conducted to evaluate and measure the performance of data stores, machine to machine interaction mechanisms and realtime processing systems.

5.1 Experimental Setup

We performed tests on two different environments to see how the systems behave on them, as our framework needs to work on both of these environments. Experiments were conducted on the default configuration provided by the systems for the Raspberrypi and experiments were conducted on virtual machine to fine tune the optimization. 1) Raspberrypi modelB 512 MB RAM. 2) Virtual machine with 3.5 GB RAM, dual core processor and 70GB hard drive. Data was loaded into MongoDb using mongoimport command, Hbase using MapReduce, Sqlite using .import command and PostgreSQL using COPY command.

5.2 Experimental Dataset

For our experiments we used temperature data of Athens, GA acquired using the weather underground API. The Weather underground is a weather forecasting system, which provides historoical as well as current weather data based on 34,000+ personal weather stations. It also gives users the option to switch to view the forecasts generated from the National Weather Services National Digital Forecast Database (NDFD) or from their personal weather stations.

5.3 Experiments on data stores:

Our framework would require a data store to store and query data in near realtime. We would query the data store more often than we write to it. Hence we would require a data store which processes random queries in less time. Experiments have been performed on PostgreSQL, Sqlite, MongoDb and HBase with index. Below are the various test cases performed on these data stores.

The Experiment shown in Figure 5.1 on page 24 was conducted to show how MongoDb, Hbase, PostgreSql and Sqlite behave while querying a single record which has been indexed. The X-axis shows the number of documents (n) present in the data store and the y-axis shows the time (ms) it took to query one record from n records. The results show that PostgreSQL and SQLite outperform MongoDb and Hbase when there are less records in the database and when there are more records in the database, MongoDb performs well.

Experiment shown in Figure 5.2 on page 25 was conducted to show how PostgreSql and Sqlite behave while querying a single record which has been indexed on a Raspberry Pi. The x-axis shows the number of documents (n) present in the data store and the y-axis shows

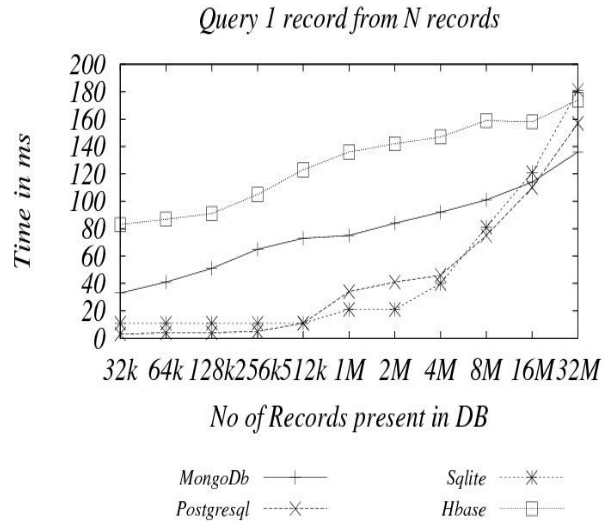

Figure 5.1: Random Read operation on various data stores on VM

the time (ms) it took to query one record from n records. The results show that Sqlite is slower than PostgreSql.

Experiment shown in Figure 5.3 on page 25 was conducted to show how MongoDb, Hbase, PostgreSql and Sqlite behave while querying for a range of records which has been indexed. The x-axis shows the number of documents (n) present in the data store and the y-axis shows the time (ms) it took to query 100 records from n records. The results show that Sqlite, PostgreSql and MongoDb perform well when there are less records in the data store. Eventually, as the size of the data store increases MongoDb performs better. MongoDb outperforms Sqlite and Postgresql because they are not meant for large volume of data. It is clear that both Sqlite and PostgreSql do not perform well after the data store reaches a specific threshold capacity.

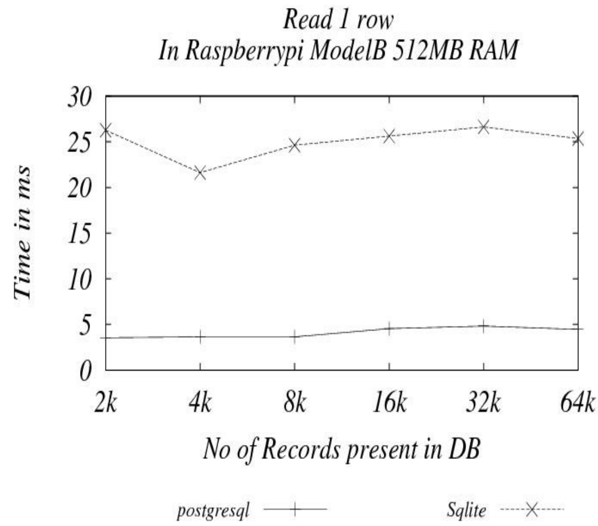

Figure 5.2: single random read operation on various data stores on RaspberryPi

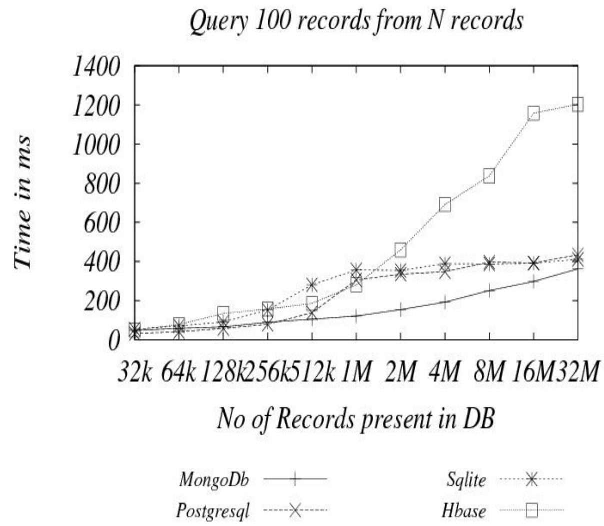

Figure 5.3: Random Read operations on various data stores on VM

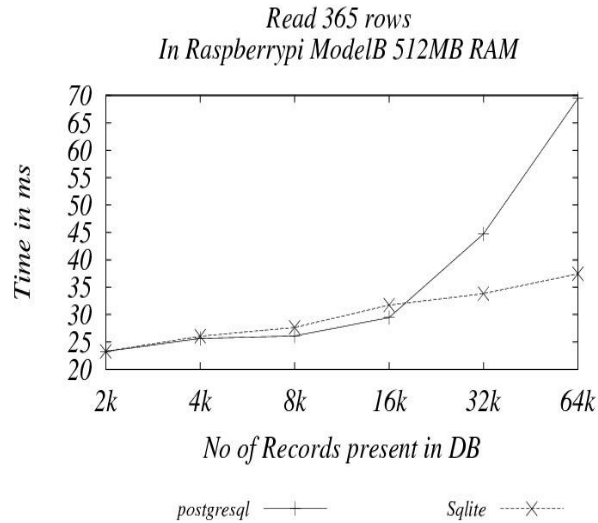

Figure 5.4: Random Read operations on various data stores on RaspberryPi

The experiment shown in Figure 5.4 on page 26 was conducted to show how PostgreSql and Sqlite behave while querying 365 records which has been indexed on a Raspberry Pi. The x-axis shows the number of documents (n) present in the data store and the y-axis shows the time (ms) it took to query one record from n records. The results show that PostgreSql performs better than Sqlite when there are less records in the data store. Eventually as the size of the data store increases Sqlite performs better.

Experiment shown in Figure 5.5 on page 27 was conducted to show how PostgreSql and Sqlite behave while writing a large number of records on a Raspberry Pi. X-axis shows the number of records being written into the data store and y-axis shows the time (ms) it took to write n records. The results show that PostgreSql performs better than Sqlite.

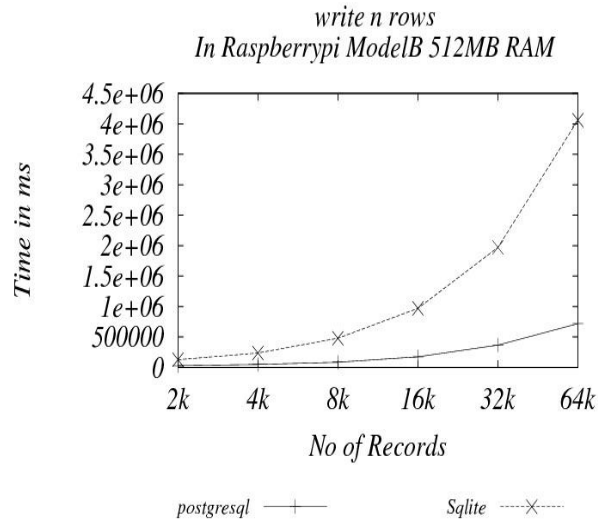

Figure 5.5: Bulk Write operations on various data stores on RaspberryPi

Experiment shown in Figure 5.6 on page 28 was conducted to show how MongoDb, Hbase, PostgreSql and Sqlite behave while writing a single record. The x-axis shows the number of records present in the data store and the y-axis shows the time (ms) it took to write 1 record. The results are pretty much similar to that of read operations.

Experiment shown in Figure 5.7 on page 28 was conducted to show how PostgreSql and Sqlite behave while writing a single record. The x-axis shows the number of records present in the data store and the y-axis shows the time (ms) it took to write 1 record. The results show that PostgreSql is much better than Sqlite.

Experiment shown in Figure 5.8 on page 29 was conducted to show how MongoDb, Hbase, PostgreSql and Sqlite behave while reading a complete database. The x-axis shows the

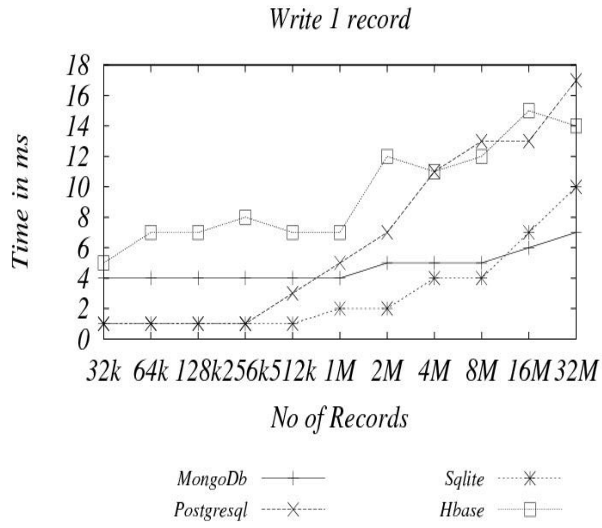

Figure 5.6: Write operation on various data stores on VM

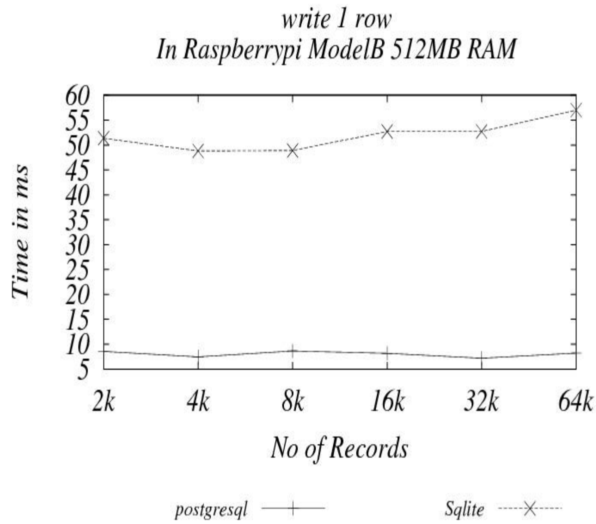

Figure 5.7: Write operation on various data stores on RaspberryPi

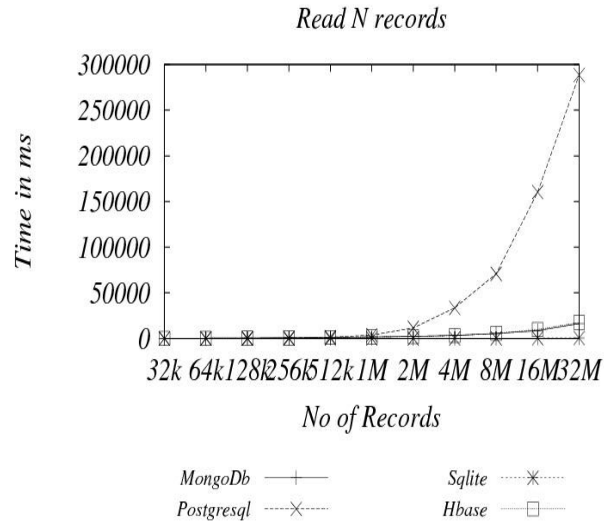

Figure 5.8: Bulk Read operations on various data stores

number of records present in the data store and the y-axis shows the time(ms) it took to read n records present in the database. Sqlite is faster than all the other data stores, this is because Sqlite writes a complete database into a single disk file. Hence it is faster when you read all the records of a database from a single disk file.

Experiment shown in Figure 5.9 on page 30 was conducted to show how PostgreSql and Sqlite behave while reading all records from a database. The x-axis shows the number of records present in the data store and the y-axis shows the time (ms) it took to read all n records from the database. The results show that Sqlite is much better than PostgreSql, this is because Sqlite writes a complete database into a single disk file. Hence it is faster when you read all the records of a database from a single disk file.

From the results we tend to see that when a random query operation is performed on

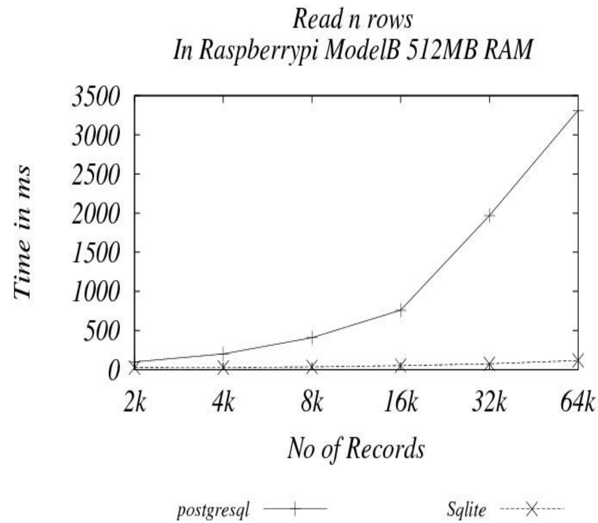

Figure 5.9: Bulk Read operations on various data stores

data stores such as MongoDb, Hbase, PostgreSql and Sqlite, all data stores performed well. However in a real case scenario, we might not always query for a single record. So we then performed tests by randomly querying 365 records from all the data stores. Sqlite and PostgreSql out performed MongoDb when there are less records in the data store. However, as the size of the data store grew, their performance declined considerably. Similarly when a random write operation is performed, MongoDb out performed Hbase, PostgreSql and Sqlite. We think that MongoDb is a suitable data store to store sensor data for our framework.

5.4 Experiments on machine to machine interaction mechanisms:

Our framework relies heavily on machine to machine interactions as there are several components which communicate via SDQ-ML within the framework. The framework requires a large number of concurrent requests to be processed. Hence we conducted tests keeping in mind the concurrent users and the number of requests processed per second.

Experiment shown in Figure 5.10 and Figure 5.11 on page 32 and page 32 was conducted to show how much time Apache Thrift and webservices take to respond to a request. The x-axis shows the number of concurrent users that are requesting for data and the y-axis shows the number of requests responded per second. From the results it is clear that Apache thrift is faster than websevices, this is because it uses remote procedural calls using TCP/IP sockets for communications and webservices uses HTTP. TCP provides communication services at an intermediate level between an application program and the IP (Internet Protocol), whereas, in HTTP there are a series of sessions in which the client sends a request and the server responds to the client. We think that Thrift is best suited for our framework.

5.5 Experiments on Realtime Processing system:

Our framework requires a realtime processing system for performing aggregate operations on the data streams. The experiments shown below are conducted by Berkeley university in 100 streams of data on 100 EC2 instances with 4 cores each.

Experiment shown in Figure 5.12 on page 33 show how Spark and Hadoop perform on a logistic regression example. The graph compares the running time per iteration of this Spark program against a Hadoop implementation on 100 GB of data on a 100-node cluster. Spark

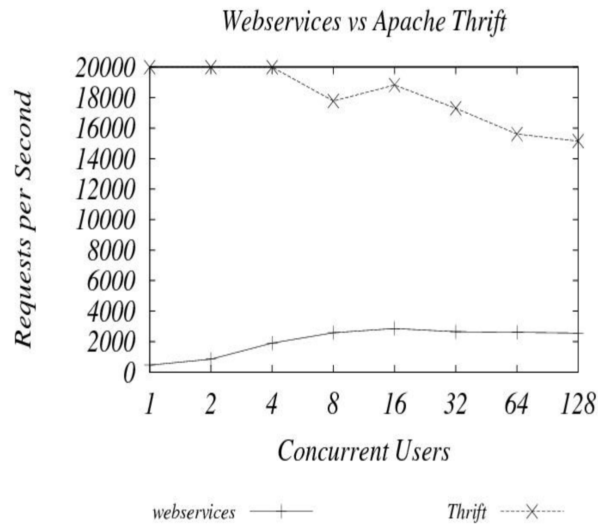

Figure 5.10: Thrift vs Web services on VM

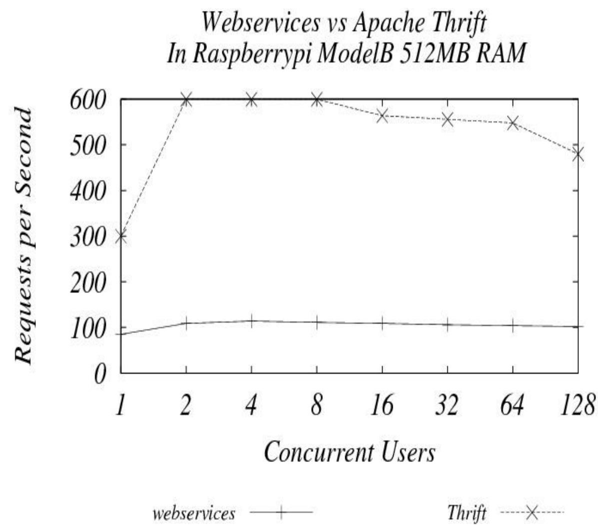

Figure 5.11: Thrift vs Web services on RaspberryPi

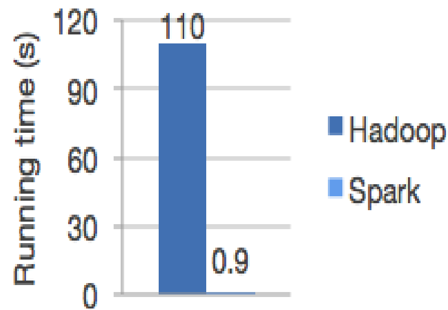

Figure 5.12: Logistic Regression on Spark and Hadoop

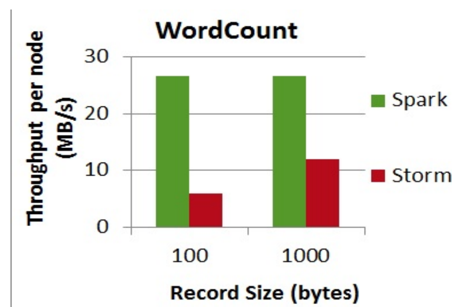

Figure 5.13: Word Count on Spark and Storm

is much faster than Hadoop because of its in memory caching capability provided through RDD.

Experiment shown in Figure 5.13 on page 33 show how Spark and Storm perform on a streaming word count example. The x-axis shows the size of individual records for which word count operation is being performed and the y-axis shows the throughput, i.e, amount of data processed per second for a given node.

Experiment shown in Figure 5.14 on page 34 show how Spark and Storm perform on a grep example. The x-axis shows the size of individual records for which grep operation is being performed and the y-axis shows the throughput. From the results it is clear that Spark performs better than Storm and Hadoop. Hence, we think that Spark is best suited for our

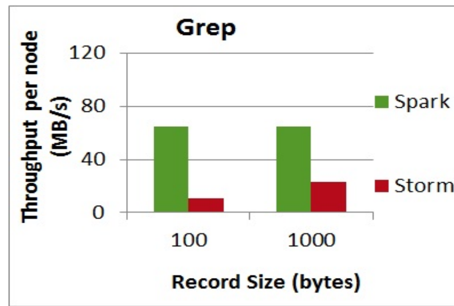

Figure 5.14: Grep on Spark and Storm

framework.

Chapter 6

Related work

SML [16] gives us an idea on how to develop a modeling language for cloud based framework. Sensor Model Language (SML) may seem like a good choice at first glance. However, SML does not provide tags for all our needs. For instance, SML neither has tags to annotate the sensor data nor to specify the requirements of the domain applications.

Wireless sensor networks (WSN), IrisNet can, in some ways, be considered as predecessors to our vision of federated sensor services. In a WSN, sensors use ad-hoc network of routers to communicate with a local monitoring base station [50, 51]. Researchers have studied various issues involved in designing WSNs including architecture, medium access control (MAC) layer protocols, routing, node localization, clock synchronization, energy management and security [50, 51, 47]. There are many works on developing operating systems, databases and other applications on sensor nodes [49, 48]. However, WSNs are not designed for sharing/trading sensor feeds in a geographically distributed setting. Our vision for a sensor service cloud is to incorporate WSNs. IrisNet envisions a global sensor web similar to federated sensor services by focusing on data collection and query answering.

Internet of Things (IOT) [46] is used to integrate real-world things with the Internet or Web. Examples for such things would be household appliances, embedded and mobile devices, smart sensing devices, etc. Often, the user interaction might take place through a cell phone acting as the mediator within the triangle of human, thing, and Internet. The application fields of the Internet of Things are influenced by the idea of ubiquitous computing. They range from smart shoes posting your running performance online, to insurance (e.g. car insurance costs based on the actually driven miles).

The Web of Things [45] can be seen as an evolvement of the Internet of Things. It leverages existing web protocols as a common language for real objects to interact with each other. HTTP is used as an application protocol. Things are addressed by URLs and their functionality is accessed through the well-defined HTTP operations (GET, POST, PUT, DELETE). Web of Things applications follow the REST standard.

Chapter 7

Conclusion

In the coming years the growth of sensor devices will be massive and accordingly the size of the data will increase exponentially. Therefore, we came up with tools and technologies necessary for processing such large volumes of data using a big data architecture. Our main objective is to provide querying capabilities based on the DQ requirements to the domain applications from large numbers of heterogeneous sensors.

Towards the end, we discussed the design requirements of a federated sensor service. We, then provide the tools and technologies which we think are a better fit for federated sensor services. A key aspect of our framework is that DQ is a first-class design artifact that is pervasive throughout the framework. We presented a unique DQ-enabled XML-based markup language for not only annotating sensor feeds but also for domain applications to specify their sensor feed requirements. Furthermore, we present a detailed analysis of the benefits and limitations of well-known big data techniques such as MongoDB, CouchDb, Hbase, Basex, Spark, Storm and Thrift in addressing the challenges involved in building federated sensor services.

Bibliography

[1] Sheth A and Thomas C and Mehra P. (2010). *Continuous Semantics to Analyze Real-Time Data*, Publisher; IEEE **27**, pp.84–89.

[2] Jeffrey Dean and Sanjay Ghemawat (2008). *MapReduce: simplified data processing on large clusters*, Publisher; Communications of the ACM

[3] Moraru, A. and Mladenic, D. (2012). *Information Technology Interfaces (ITI)*, Publisher; International Conference **25–28**, pp.155–160.

[4] Chen-Khong Tham andRajkumar Buyya(2003). *SensorGrid: Integrating sensor networks and grid computing*, Publisher; CSI communications

[5] I. Akyildiz and M. Vuran, D. (2010). *Wireless sensor networks*, Publisher; Wileye **4**, pp.155–160.

[6] Sheth A (2008). *Semantic Sensor Web*, Publisher; IEEE **12**, pp.78–83.

[7] Mohammad Mehedi Hassan, Biao Song, Eui-Nam Huh (2009). *A framework of sensor-cloud integration opportunities and challenges*, pp.618–626

[8] J. Stankovic, D. (2008). *Wireless sensor networks* **41**

[9] S. Nath et al. (2003). *Irisnet: An architecture for internet-scale sensing services*

[10] Apache SOLR , Website; http://wiki.apache.org/solr/SolrPerformanceData

[11] E. Wilde (2009). *Making sensor data available using web feeds*, Publisher; IPSN

[12] Eric Brewer (2010). *A certain freedom: thoughts on the CAP theorem*, Publisher; Principles of Distributed Computing

[13] Randy Abernethy. (2013). *The Programmer's Guide to Apache Thrift*, Publisher; manning publication co.

[14] Lars George. (2011). *HBase: The Definitive Guide*, Publisher; O'Reilly Media.

[15] Kristina Chodorow. (2013). *MongoDB: The Definitive Guide*, Publisher; O'Reilly Media.

[16] OpenGeospatialConsortium(OGC). *Sensor Model Language(SML)*, Website; http://www.opengeospatial.org/standards/sensormlas.

[17] XIVELY- Internet of Things Platform. , Website; http://www.opengeospatial.org/standards/sensormlas.

[18] Big Data , Website; http://strata.oreilly.com/2012/01/what-is-big-data.html

[19] MongoDb - NOSQL Database , Website; http://www.mongodb.com/leading-nosql-database

[20] HBase - Data Store , Website; https://hbase.apache.org/

[21] Hive - data warehouse software , Website; http://hive.apache.org/

[22] Pig - data warehouse software , Website; https://pig.apache.org/

[23] postgresql - Relational Database Management System , Website; http://www.postgresql.org/

[24] mysql - Relational Database Management System , Website; http://www.mysql.com/

[25] CouchDb - NOSQL Database , Website; http://couchdb.apache.org/

[26] sedna - XML Database , Website; http://www.sedna.org/

[27] BaseX - XML Database , Website; http://basex.org/

[28] Thrift - RPC , Website; http://thrift.apache.org/

[29] resteasy - RESTful Web Services , Website; http://www.jboss.org/resteasy/

[30] STORM - Distributed Realtime Stream Processing , Website; http://storm-project.net/

[31] SPARK -Lightning-fast cluster computing , Website; http://spark.apache.org/

[32] Azure DataMarket , Website; http://datamarket.azure.com/

[33] Pottie, G.J. (1998). *Wireless sensor networks*, Publisher; IEEE **27**, pp.139 – 140.

[34] Infochimps , Website; http://www.infochimps.com/

[35] International Data Corporation , Website; https://www.idc.com/

[36] Wireless World Research organization , Website; http://www.wwrf.ch/

[37] Central Nervous System for the Earth , Website; http://www.hpl.hp.com/news/2009/oct-dec/cense.html

[38] wunderground - Repository for weather Data , Website; http://www.wunderground.com/

[39] Microsft SQL Server , Website; http://www.microsoft.com/en-us/server-cloud/products/sql-server

[40] IBM DB2 , Website; http://www-01.ibm.com/software/data/db2/

[41] Oracle , Website; http://www.oracle.com/us/products/database

[42] Xpath , Website; http://www.w3.org/TR/xpath/#section-Introduction

[43] Xquery , Website; http://www.w3.org/XML/Query/

[44] Hadoop , Website; http://hadoop.apache.org/

[45] Guinard, D. and Trifa, V. (2009). *Towards the Web of Things: Web Mashups for Embedded Devices*,publisher;International World Wide Web Conference

[46] Gershenfeld, N.; Krikorian, R. and Cohen, D. (2004). *The Internet of Things.*, Publisher; Scientific American **291**, pp.76-81.

[47] D. H. Kim (2010). *Sensloc: sensing everyday places and paths using less energy,*, Publisher; SenSys

[48] S. Madden (2005). *Tinydb: an acquisitional query processing system for sensor networks,*, Publisher; ACM Trans. Database Syst. **30**

[49] S. Madden (2010). *Database abstractions for managing sensor network data,*, Publisher; Proceedings of the IEEE **98**

[50] I. Akyildiz and M. Vuran(2010). *Wireless sensor networks*, Publisher; Wiley, 2010 **4**

[51] J. Stankovic(2008). *Wireless sensor networks,*, Publisher; Computer, 2010 **41**

Appendix A

SAMPLE SDQML

Listing A.1: SDQ-ML

```
<?xml version="1.0"?>
<!--Sensor Data Quality Model Language-->
<SDQML>
<!--Sensor Feed Description Language-->
<SFDL>
<sensor>
<id>YSI_RG_0001</id>
<name>light</name>
<location>
<latitude>90.80</latitude>
<longitude>90.80</longitude>
</location>
<duration start="1" end="2"/>
<!-- No of readings per hour-->
<frequency>30</frequency>
</sensor>
<!--its all the quality parameters-->
<dqmetric>
```

```
19 <sensoraccess></sensoraccess>
20 <accuracy>80</accuracy>
21 <error_rate>90</error_rate>
22 <!--in secs-->
23 <timeliness>20</timeliness>
24 <!--response time of a given mote or group in secs-->
25 <responsetime>1</responsetime>
26 <validtime>80</validtime>
27 <satisfaction>90</satisfaction>
28 <trustworthy>78</trustworthy>
29 <avalibility>65</avalibility>
30 <!--Equipment Quality of a sensor MOTE or group-->
31 <equipqual>78</equipqual>
32 <!--Mesh Network Quality-->
33 <netqual>65</netqual>
34 <!--Quality of an individual MOTE or group-->
35 <motequal>55</motequal>
36 <!--The quality of data usage. E.g. did it delivery what the user
     requested based on number of users requesting that feed-->
37 <usability>90</usability>
38 <latency>5</latency>
39 <uniqueness>90</uniqueness>
40 <demand>65</demand>
41 </dqmetric>
42 <description>
43 <storage>databasename</storage>
44 <location>192.81.0.0</location>
45 </description>
46 </SFDL>
47 <!--Sensor User Request Language-->
48 <SURL>
```

```
49 <sensor>
50 <id>YSI_RG_0001</id>
51 <name>light</name>
52 <location>
53 <latitude>90.80</latitude>
54 <longitude>90.80</longitude>
55 </location>
56 <duration start="2012-05-30T09:00:00" end="2014-05-30T09:00:00"/>
57 <!-- No of readings per hour-->
58 <frequency>30</frequency>
59 </sensor>
60 <request>
61 <range from="10.0" to="14.0"/>
62 <aggregate>sum</aggregate>
63 </request>
64 </SURL>
65 <dqmetric condition=">" match="all,few,atleast" parameters="">
66 <sensoraccess></sensoraccess>
67 <accuracy>80</accuracy>
68 <error_rate>90</error_rate>
69 <!--in secs-->
70 <timeliness>20</timeliness>
71 <!--response time of a given mote or group in secs-->
72 <responsetime>1</responsetime>
73 <validtime>80</validtime>
74 <satisfaction>90</satisfaction>
75 <trustworthy>78</trustworthy>
76 <avalibility>65</avalibility>
77 <!--Equipment Quality of a sensor MOTE or group-->
78 <equipqual>78</equipqual>
79 <!--Mesh Network Quality-->
```

```
80 <netqual>65</netqual>
81 <!--Quality of an individual MOTE or group-->
82 <motequal>55</motequal>
83 <!--The quality of data usage. E.g. did it delivery what the user
      requested based on number of users requesting that feed-->
84 <usability>90</usability>
85 <latency>5</latency>
86 <uniqueness>90</uniqueness>
87 <demand>65</demand>
88 </dqmetric>
89 </SDQML>
```

Appendix B

Experiments Data

Table B.1: Random Read: Query 1 record from MongoDb

Number of Records in Database	Test Cases (Time in ms)									
	Test1	**Test2**	**Test3**	**Test4**	**Test5**	**Test6**	**Test7**	**Test8**	**Test9**	**Test10**
3200	36	32	35	28	33	38	34	29	33	36
64000	44	41	43	38	42	47	40	42	41	39
128000	57	53	49	52	54	48	49	51	53	49
256000	68	63	65	66	61	68	69	65	70	62
512000	77	74	73	75	76	71	64	69	75	78
1000000	81	70	74	77	74	73	78	71	79	75
2000000	82	84	94	86	82	79	88	82	85	80
4000000	101	94	89	93	90	88	95	89	88	96
8000000	106	96	99	110	108	98	97	102	98	101
16000000	112	121	116	114	104	118	110	118	121	111
32000000	145	137	129	133	138	128	139	140	145	133

Table B.2: Random Read: Query 100 records from MongoDb

Number of Records in Database	Test Cases (Time in ms)									
	Test1	**Test2**	**Test3**	**Test4**	**Test5**	**Test6**	**Test7**	**Test8**	**Test9**	**Test10**
32000	49	46	45	51	47	46	48	49	46	45
64000	59	58	56	57	58	59	60	57	59	57
128000	68	65	67	66	67	68	66	67	65	65
256000	91	89	88	87	90	91	90	89	88	89
512000	109	106	103	102	105	107	106	104	105	106
1000000	126	120	122	118	124	125	119	121	121	119
2000000	159	156	152	154	151	159	156	154	150	153
4000000	196	193	191	192	194	193	191	190	188	193
8000000	252	255	250	249	251	252	253	251	250	251
16000000	301	296	298	299	302	301	297	295	299	297
32000000	357	360	361	365	363	362	363	364	362	363