A CLOUD BASED ARCHITECTURE FOR DISTRIBUTED REAL TIME PROCESSING OF CONTINUOUS QUERIES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS INSTITUTE
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MERT ONURALP GÖKALP

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
INFORMATION SYSTEMS

September, 2015

A CLOUD BASED ARCHITECTURE FOR DISTRIBUTED REAL TIME PROCESSING OF CONTINUOUS QUERIES

Submitted by **Mert Onuralp GÖKALP** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems, Middle East Technical University** by,

Prof. Dr. Nazife Baykal ______________________________
Director, Informatics Institute

Prof. Dr. Yasemin Yardımcı Çetin ______________________________
Head of Department, Information Systems

Assoc. Prof. Dr. Altan Koçyiğit ______________________________
Supervisor, Information Systems, METU

Examining Committee Members:

Assoc. Prof. Dr. Banu Günel ______________________________
Information Systems, METU

Assoc. Prof. Dr. Altan Koçyiğit ______________________________
Information Systems, METU

Assist. Prof. Dr. P. Erhan Eren ______________________________
Information Systems, METU

Assist. Prof. Dr. Ayça Tarhan ______________________________
Computer Engineering, Hacettepe University

Assoc. Prof. Dr. Alptekin Temizel ______________________________
Modeling and Simulation, METU

Date: 04.09.2015

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name and Surname :	**Mert Onuralp Gökalp**
Signature :	**____________________**

ABSTRACT

A Cloud Based Architecture for Distributed Real
Time Processing of Continuous Queries

Gökalp, Mert Onuralp
M.S. Department of Information Systems
Supervisor: Assoc. Prof. Dr. Altan Koçyiğit

September 2015, 57 pages

The technological advancements in Internet of Things (IoT) domain have enabled us to reshape the physical world through smart devices, sensors and actuators. The data collected by IoT devices has become a valuable asset to extract knowledge about the environment and other nearby devices. Existing IoT applications mostly store collected data in a central server and allow users to query stored data to notice and react to changes in the environment. Usually cloud and big data technologies are utilized in those applications for scalability. Nevertheless, the responsiveness of such IoT applications is limited due to the use of polling based queries. In this thesis, we primarily focus on the problem of specifying a generic and scalable architecture to process a multitude of continuous queries in real time, respond to events and notify users in a timely manner. For this purpose, we propose a data-flow based query definition model to allow users create flexible queries. We devise a centrally managed distributed infrastructure based on the state of the art big data technologies to execute the continuous queries over streaming data rather than storing and frequently querying the data collected. A prototype has been implemented to demonstrate the applicability and to evaluate the scalability of the proposed approach.

Keywords: Cloud Computing, Internet of Things, Stream Processing, Big Data, Continuous Query, Distributed Computing.

ÖZ

Dağıtık Gerçek Zamanlı Sürekli Sorguları İşlemek için
Bulut Tabanlı bir Mimari

Gökalp, Mert Onuralp
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Doç. Dr. Altan Koçyiğit

Eylül 2015, 57 sayfa

Nesnelerin İnterneti(Nİ) alanında gerçekleşen teknolojik gelişmeler bize fiziksel dünyayı akıllı aletler, algılayıcılar ve eyleyiciler aracılığıyla yeniden şekillendirebilme imkanı sağlamıştır. Nİ cihazlarıyla toplanan veriler bulunulan ortam ve yakınlardan bulunan diğer cihazlar hakkında bilgi edinmek için önemli bir varlık haline gelmiştir. Mevcut Nİ uygulamaları genellikle verileri merkezi bir sunucuda toplar ve kullanıcıların ortamdaki değişiklikleri fark etmeleri ve tepki vermeleri için toplanan veriler üzerinde sorgu yapmalarına imkan vermektedir. Ölçeklenebilirlik için bu uygulamalarda genellikle bulut ve büyük veri teknolojileri kullanılmaktadır. Yine de bu tarz Nİ uygulamalarının tepki verimliği tarama tabanlı sorgulama kullanımı nedeniyle sınırlıdır. Bu tezde, birincil olarak, çok sayıda sürekli sorguyu gerçek zamanlı işlemek, zamanlıca olaylara tepki vermek ve kullanıcıları uyarmak için genel-geçer ve ölçeklenebilir bir mimari tanımlama problemi üzerine odaklanıyoruz. Bu amaçla, kullanıcıların esnek sorgular tanımlayabilmesi için veri akışı tabanlı bir sorgu tanımlama modeli öneriyoruz. Verileri saklamak ve saklanmış verileri sürekli sorgulamak yerine sürekli sorguları akan veriler üzerinde işlemek için güncel büyük veri teknolojilerine dayalı, merkezi yönetimli dağıtık bir altyapı tasarlanmaktadır. Önerilerin yaklaşımının uygulanabilirliğini göstermek ve ölçeklenebilirliğini ölçmek için örnek bir uygulama gerçekleştirilmiştir.

Anahtar Kelimeler: Bulut Bilişim, Nesnelerin İnterneti, Akan Veri İşleme, Büyük Veri, Sürekli Sorgu, Dağıtık Hesaplama

To my parents,
Saadet and *Hidayet GÖKALP*

ACKNOWLEDGEMENT

First and foremost, I would like to express my sincere appreciations to my supervisor Assoc. Prof. Dr. Altan Koçyiğit. He has given me warm supports, valuable guidance, generous advice, time and shown great patience through emails than I ever would expect from an advisor.

I would like to thank Assist. Prof. Dr. P. Erhan Eren for his support, suggestions, comments and for sharing his knowledge through this research.

I wish to thank my sister and colleague Ebru Gökalp for her patience and sharing her experience and knowledge while preparing this thesis study.

I would like to thanks my sister Emel Gökalp, my brother Cenk Gökalp for their guidance and advices throughout my life, and my nephews Gökdeniz and Hidayet Gökalp for making my life more enjoyable during this study.

I wish to thank Özeren Bulut and Selin Çoban for their help and encouragement during the study.

Finally, I would like to express my very special gratitude to my parents, Saadet and Hidayet Gökalp for their exceptional patience, love, encouragement and endless support.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

ACKNOWLEDGEMENT vii

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ACRONYMS xiv

CHAPTERS

INTRODUCTION 1

BACKGROUND AND RELATED WORKS 3

2.1 Related Works 3

2.2 Cloud Computing 5

2.3 Big Data 5

2.4 Storm 6

2.5 HBase 8

2.6 Kafka 8

2.7 Node Red 9

2.8 Zookeeper 10

CONTINUOUS QUERIES 11

3.1 The Proposed Continuous Query Model 11

3.2 Query Elements 12

3.3 Motivating Examples 13

3.3.1 Traffic Lights Management System for Four Leg Intersection Roads .. 13
3.3.2 Home Automation System 15
3.3.3 Sport Tracker System 15
ARCHITECTURE 17
4.1 System Architecture 17
4.1.1 Query Generation and Representation 19
4.1.2 Query Processing 20
4.1.3 Data Distribution 20
4.1.4 Cloud Application Server 21
4.2 Query Implementation 22
4.2.1 Sensor 22
4.2.2 Average 22
4.2.3 Sum 23
4.2.4 MovingAverage 23
4.2.5 Max 24
4.2.6 Min 24
4.2.7 CompareSensors 25
4.2.8 Threshold 25
4.2.9 Groovy Script 26
4.2.10 Twitter 27
4.2.11 HBase 27
4.2.12 Web Service 27
4.2.13 Socket 28
PROTOTYPE IMPLEMENTATION 29

5.1 Query Generation Module 29
5.2 Data Distribution Module 31
5.3 Query Processing Module 32
5.4 Cloud Application Server Module 34
5.4.1 Deploying a Topology 35
EVALUATIONS 37
6.1 Experimental Setup 37
6.2 Scenarios 38
6.3 Scalability Experiments & Results 43
6.3.1 One Slave Node 43
6.3.2 FourSlave Nodes 45
6.4 Comparison of Alternative Data Distribution Methods 47
CONCLUSION 51
REFERENCES 55

LIST OF TABLES

Table 4.1. Average Node Parameters 22

Table 4.2. Sum Node Parameters 23

Table 4.3. Moving Average Node Parameters 23

Table 4.4. Max Node Parameters 24

Table 4.5. Min Node Parameters 24

Table 4.6. CompareSenssors Node Parameters 25

Table 4.7. Threshold Node Parameters 26

Table 4.8. Example Groovy Script 27

Table 4.9. Twitter Node Parameters 27

Table 4.10. Web Service Node Parameters 28

Table 4.11. Socket Node Parameters 28

Table 5.1. Storm Bolt Methods 33

Table 5.2. HBase Row Key Designs 33

Table 6.1. Queries for Scenario 1 39

Table 6.2. Queries for Scenario 2 41

Table 6.3. Queries for Scenario 3 42

LIST OF FIGURES

Figure 2-1. Storm Abstractions 7
Figure 2-2. Storm Component Diagram 7
Figure 2-3. High Level Architecture of Kafka 9
Figure 2-4. Basic Node Red Flow 10
Figure 3-1. The Query Model 12
Figure 3-2. Sample Query 12
Figure 3-3. The Four-Leg Intersection Road Sketch 14
Figure 3-4. Traffic Lights Management System Query 14
Figure 3-5. Home Automation System Query 15
Figure 3-6. Sport Tracker System Query 16
Figure 4-1. System Architecture 19
Figure 4-2. Data Distribution Module 21
Figure 4-3. Query Object Class Diagram 26
Figure 5-1. Query Generation Module Package Diagram 29
Figure 5-3. Node Red Visual Interface 31
Figure 5-4. Data Distribution Module Package Diagram 31
Figure 5-5. Query Processing Module Package Diagram 32
Figure 5-6. Cloud Application Server Package Diagram 34
Figure 5-7. Entity Relationship Diagram 35
Figure 5-8. Sample Query 36
Figure 5-9. Storm Topology Graph of Sample Query 36

Figure 6-1. System Deployment Diagram 38
Figure 6-2. Traffic Lights Management System for Four-Leg Intersection Roads ... 39
Figure 6-3. Home Automation System 40
Figure 6-4. Sport Tracker System 42
Figure 6-5. Average Latency with One Slave Node for Scenario 1 43
Figure 6-6. Average Latency with One Slave Node for Scenario 2 44
Figure 6-7. Average Latency with One Slave Node for Scenario 3 44
Figure 6-8. Average Latency with Four Slave Nodes for Scenario 1 45
Figure 6-9. Average Latency with Four Slave Nodes for Scenario 2 45
Figure 6-10. Average Latency with Four Slave Nodes for Scenario 3 46
Figure 6-11. Average Latency for the Mixture of Scenarios 47
Figure 6-12. Performance Comparison of Slave Nodes 47
Figure 6-13. Straightforward Approach 48
Figure 6-14. Proposed Approach 49
Figure 6-15. Messaging System Test Results-1 50
Figure 6-16. Messaging System Test Results-2 50

LIST OF ACRONYMS

API Application Programming Interface

CAS Cloud Application Server

CEP Complex Event Processing

CQ Continuous Query

CQL Continuous Query Language

EP Event Processing

ER Entity Relationship

GUI Graphical User Interface

HDFS Hadoop Distributed File System

IaaS Infrastructure as a Service

IoT Internet of Things

JVM Java Virtual Machine

LTS Long Term Support

M2M Machine-to-Machine

MR Map-Reduce

PaaS Platform as a Service

SaaS Software as a Service

UI User Interface

CHAPTER 1

INTRODUCTION

The Internet of Things (IoT) concept has attracted significant research interest as a result of technological advancements and innovations in smart-device, smart-sensor and actuator technologies. According to Gartner [1], the IoT devices (excluding smart phones, tablets and PCs) will grow up to 26 billion units by the year 2020.

The IoT concept refers to the network formed by smart objects that can connect to the Internet and communicate with each other over the Internet. Hence, we are able to query the physical world through smart devices. On the other hand, the network of such smart objects can generate a huge number of data streams. Thus, the IoT concept comes with a big issue, named Big Data.

The Big Data term is generally used to describe the exponential growth and availability of data, both structured and unstructured. There are mainly three common aspects of Big Data applications: Volume, Velocity and Variety. The big network of smart objects is able to produce enormous amount of information per second and with different presentation formats. Thus, the architecture of IoT applications should consider these three aspects of Big Data.

The IoT applications also need a fast and scalable architecture to process and store this big data in an effective manner. In general, a single server is not enough to store and process massive data. In other words, performing these operations in a single centralized server infrastructure is not always efficient. Hence, distributed computing environments turn out to be viable alternatives.

The applications in IoT domain usually produce and process continuous data streams. Moreover, these devices should rapidly adapt to changes in the environment. In order to make this possible, users should be able to define flexible rules, event and time based triggers, scripts and notifications. In the literature, there are many studies on collecting, storing and querying the potentially enormous amount of data. However, most of these studies mainly target a specific use case and they are primarily based on querying stored data. In order to make those approaches scalable, the state-of-the-art big data technologies are being utilized. Nevertheless, the responsiveness of such IoT applications is limited due to polling based queries employed.

In this thesis study, we focus primarily on the continuous queries and the software architectures to process a multitude of continuous queries over data streams originated from IoT devices and to respond to events and notify users in real time. We proposed a data-flow based continuous query definition model that allow users to define flexible queries. The continuous queries may consist of some statistical computations, rules, event/time based triggers, Groovy [35] scripts, notifications and/or data storage. We also develop a centrally managed distributed infrastructure which utilizes state-of-the-art cloud computing and big data technologies including Storm [3] for data processing in real-time, HBase [4] for data storage, Kafka [5, 6] for data distribution, Node Red [7] for query definition and Zookeeper [8, 9] to ensure synchronization among processing nodes.

In order to demonstrate the applicability of the proposed architecture we implemented a prototype. We conducted several experiments on our prototype implementation to evaluate the scalability of the architecture.

This thesis consists of seven chapters. After this introductory chapter, Chapter 2 aims to give an overview of Cloud Computing, Big data, the technologies utilized in our architecture and the related literature. Chapter 3 describes the proposed query definition model and gives motivating use case scenarios. Chapter 4 describes the devised system architecture to process multiple continuous queries in real time and explains in detail how a continuous query is created and executed within the proposed framework. In Chapter 5, the detailed information about prototype implementation is given. The experimental setup, test scenarios and the evaluation of the prototype implementation are given in Chapter 6. Finally, the concluding remarks and directions for future research are given in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORKS

This chapter provides an overview of the Continuous Queries (CQ), cloud computing and big data domains and the technologies utilized in the architecture proposed in this thesis. In section 2.1, the related works in the CQ domain is reviewed. In section 2.2, we provide an overview for cloud computing. In section 2.3, the big data concept and its general characteristics are explained. In section 2.4, Storm framework is described in detail; the features, abstractions and the areas of usage are explained. Section 2.5 describes the Hadoop [11] environment and the HBase database management system, which runs on top of the Hadoop. In section 2.6, Kafka publish/subscribe messaging system is described. In section 2.7, the notable features of Node Red are explained. In section 2.8, the Zookeeper framework is described.

2.1 Related Works

Numerous studies related to continuous queries and stream processing can be found in the literature. The research on data streaming and continuous queries is reviewed by Babcock, Brian, et al. [18]. Nevertheless, the field of real time stream processing in a distributed environment is a new field of study.

Several architectures have been proposed to process continuous queries such as Tapestry [19], OPENCQ [20], NiagaraCQ [21]. Tapestry system allows users query over append-only SQL database for filtering streams of electronic documents such as mails and news messages. This system, periodically queries over database with SQL and merges the results to produce results of continuous query. OPENCQ and NiagaraCQ focus on continuous queries over traditional database sources and thus don't deal with issues specific to streaming sensor data.

OPENCQ provides a continuous query system over persistent data sets for event-driven information delivery. NiagaraCQ differs from OPENCQ in bringing similar queries together for reducing I/O costs, avoiding unnecessary invocations and sharing computational resources.

Two recent systems, Cougar [22] and TinyDB [23] deal with query processing in sensor networks. Cougar and TinyDB are distributed query processors that run on sensor nodes with the TinyOS [38] operating system.

Aurora [24], Borealis [37] and STREAM [25] are continuous query systems over streaming data. Aurora and Borealis are workflow-oriented systems that allow user to build query plans by arranging operators and arrows. They are designed for monitoring applications to manage data streams. STREAM is an all-purpose relation-based system with an emphasis on memory management that approximates query answering. STREAM uses an SQL like language CQL [26, 27] (Continuous Query Language) which supports windowing and ordering. Both of these systems can process streaming data but they are designed as centralized systems. They do not have support for distributed infrastructures.

Rule engines can also be used to process continuous queries. There are two recent rule engines to process queries that are formed as rules such as Esper [28] and Drools [29]. Esper and Drools are specialized in Complex Event Processing (CEP). These systems analyze and filter streaming messages in real time. Esper provides a high level SQL like query language and Drools provide a Drools Rule Language which is like "when, then" sentence. Nevertheless, these systems are designed mainly for centralized streaming and thus they don't inherently support distributed processing.

Integrating sensors and cloud services have been discussed in Dash et al. [30] and Alamri et al. [31]. Both of these studies define a general architecture to connect sensors to cloud services. However, there are potential issues such as storage, authorization and scaling in this domain.

In the field of real-time stream processing, Storm, S4 and Spark Streaming [10] are the most notable frameworks that focus on large scale and low latency stream computation. Storm and S4 are specialized in stream processing but Spark Streaming uses micro-batching model to treat sequence of data as a streaming data.

Integrating big data and stream processing tools have been discussed in Rios et al. [32]. They process received sensor data by Storm before storing them to the database. The main limitation of this study is that their processing module is not query based. In other words, they process data with a predefined processing algorithm.

Lim and Babu [33] define different execution plans for continuous windowed aggregation queries. They made comparisons for centralized vs. distributed execution engines and streaming vs. repeated batch execution. However, their study is limited to picking the best execution plan for given aggregation queries.

2.2 Cloud Computing

Cloud computing is an emerging paradigm to provide hardware and software resources over Internet for third party services. The most common services of cloud computing are [34];

- *Software as a Service (SaaS):* It is a way of delivering software to many consumers as a web based application accessed over the Internet.
- *Platform as a Service (PaaS):* This service provides a development environment and an execution platform for developers over the Internet.
- *Infrastructure as a Service (IaaS):* It is the delivery of computation infrastructure over the Internet.

There is a relationship between cloud computing and big data paradigms. The big data applications need to process, analyze and store a large number of records. Thus, big data applications need huge data stores, an extensive computation power and a distributed infrastructure. Hence, cloud computing makes it easier and cheaper develop and run big data applications. The cloud computing consumers use IaaS to provide suitable infrastructures for their applications, consume the application development environment as a PaaS and/or use the SaaS to collect data. Therefore, there is a mutual advantage between cloud computing and big data.

2.3 Big Data

Big data is an ambiguous term to describe. Generally it is used to describe the exponential growth and availability of structured or unstructured data. There are three common aspects that make Big Data term more understandable; Volume, Velocity and Variety.

- *Volume:* This characteristic refers to amount of the data that is to be collected and stored. With the recent advances in sensor and machine-to-machine technologies, these devices can generate exponentially growing volume of data. An important challenge is building an infrastructure to deal with that huge data. In general, single server is not enough to store and process these data sets; fast and scalable architectures are required for this.
- *Velocity:* This characteristic refers to the speed of data generation. Today, people use social media to update them with the latest news. Statuses, tweets and etc. can change in a second. For instance, Twitter produces 80 MB of information per second that is around 8 TB per day [12]. Data velocity also refers to the amount of data that can be processed in a unit time interval. It is important that processing “in movement” data quickly enough to deal with data velocity challenge.
- *Variety:* This characteristic emphasizes that the source and the presentation form of the data are diverse, and it’s often hard to fit it into relational structures. Data can be generated by RFID tags, web-sites, GPS sensors, etc. All of these sources may generate data in a different format. Defining a

common input format in the applications and transforming data into that format is an important challenge.

Big Data applications generally use highly scalable systems to process the data in an efficient way. There are three different systems to deal with different characteristics of the Big Data; Stream Processing, Batch Processing and Micro-Batch Processing.

- Stream Processing: It is an efficient way to process high velocity data. Stream processing applications run continuously to process incoming data. The input is usually produced and delivered at run time. Storm and S4 [13] are the most notable frameworks to process streaming data.

- Batch Processing: It is an efficient way to process a high volume of data. In batch processing systems, data should be placed in a persistent storage before computation starts. On the other hand, the output of the computations is available when all of the data is processed. Thus, this processing technique is more suitable to analyze historical data. The Hadoop framework is specialized in batch processing.

- Micro-Batch Processing: Micro-batching is separating batch data into small chunks or collecting streaming data to treat as sequence of streaming data. This processing technique is generally used for machine learning algorithms. Storm Trident [14] and Spark [15] are the most notable frameworks for micro-batch processing.

2.4 Storm

Storm[3]is a distributed real time data stream-processing platform, which is available under the Apache Open Source license. In this thesis, Storm is used as the core framework to process data streams because of its five key characteristics: speed, scalability, fault tolerance, reliability and ease of operation. In order to understand how Storm processes data, we can look at its five key abstractions (illustrated in Figure 2-1):

- Tuples: A set of key-value pairs.
- Stream: An unbounded sequence of tuples.
- Spouts: The source of input data streams for topologies. Spouts can read data from external sources and also from existing topologies.
- Bolts: The processing units of topologies. Basically, a bolt processes input streams to produce output streams. An output stream can be written into a database or it can be emitted to another bolt. There may be one or more threads for each bolt.
- Topology: A network of spouts and bolts. Topologies define the application logic. In the configurations of topologies, a developer can define the number of worker threads, which execute the topology.

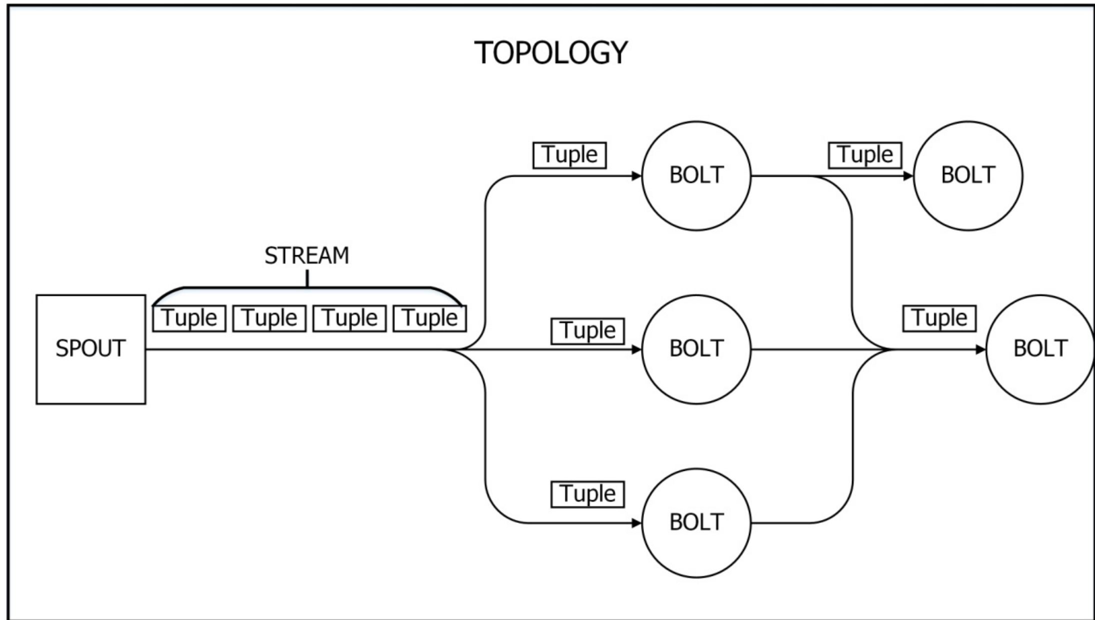


Figure 2-1. Storm Abstractions

The main components of Storm are Nimbus, Supervisor and User Interface (UI). Nimbus component is responsible for distributing topologies and the executable code of the topologies to slave nodes. Each slave node runs a Supervisor component. The supervisor starts and stops the jobs assigned by Nimbus. The synchronization between Nimbus and Supervisor is provided by Zookeeper. Zookeeper is an open source service that provides synchronization among clusters for distributed systems. It provides high availability and high throughput with low latency. Storm uses Zookeeper to ensure coordination across processes that run on different nodes. The topologies can be monitored using the UI component. The component diagram of Storm is given in Figure 2-2.

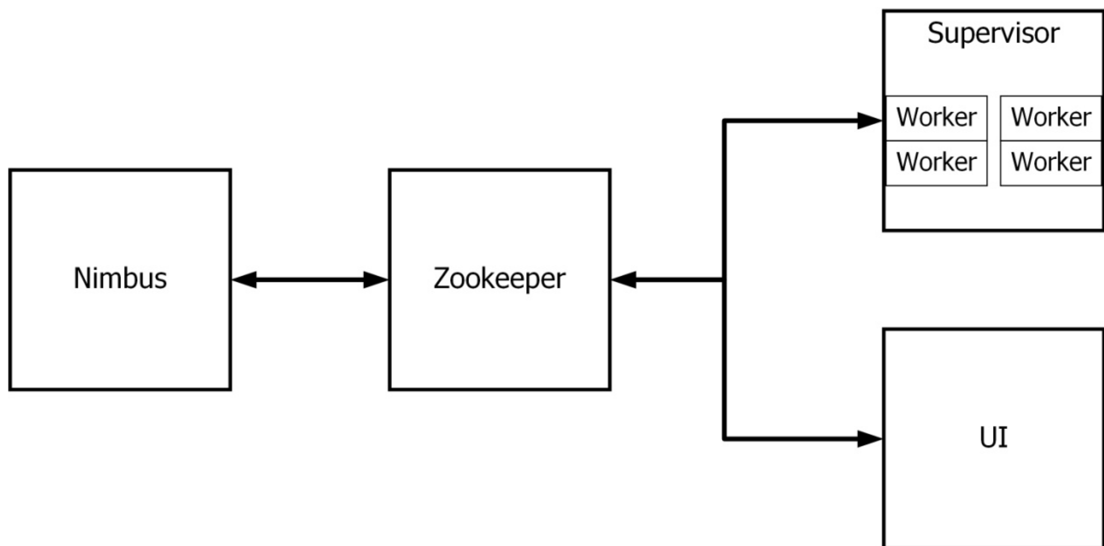


Figure 2-2. Storm Component Diagram

2.5 HBase

Hadoop is one of the most important environments to process big batch data. The key features of Hadoop are reliability, scalability and durability. Hadoop environment is designed for processing big batch data in a distributed environment setting. There are lots of tools and database management systems that run on top of Hadoop environment. HBase is one of such database management systems and it runs on the Hadoop environment.

HBase is an open source, column oriented and non-relational-NoSQL database management system that is built on Hadoop environment. HBase runs on top of the Hadoop Distributed File System (HDFS) [16], which provides scalable, replicated and persistent data storage to HBase.

HDFS keeps data in files called HFiles. HFiles contain sparse, distributed, persistent and multidimensional-sorted map, which is indexed by a row key, column key, and a timestamp. The main components of HDFS are NameNode and DataNode. NameNode is responsible for maintaining directories and files and managing data blocks of DataNode. The main data storage component is DataNode and it also handles the read/write requests for clients. HBase distributes files via HMaster process to HRegionServers. The synchronization among HRegionServers is ensured by Zookeeper.

In order to run Hadoop in a distributed manner, there are two important components: Resource Manager and Node Manager. The main node of the Hadoop is the Resource Manager. It is responsible for scheduling the jobs and distributing them to the Node Managers. The Node Manager handles launching and monitoring the assigned jobs.

The main benefits of HBase are being a fault tolerant storage, providing a flexible data model, performing near real-time read/write, allowing replication across the data center, enabling automatic load balancing of tables and providing high availability through automatic failovers.

HBase data model organizes data in tables. Within a table, data is stored rows. Data within a row is grouped by a column family. Data within a column family is addressed via its column-qualifier. A combination of a row key, a column family and a column qualifier uniquely identifies a cell. Values within a cell are versioned with timestamps. Table rows are sorted according to their row keys that serve as primary keys. Therefore, the row key design is the single most important issue to determine how the system will communicate with the HBase.

2.6 Kafka

Kafka is a distributed publish-subscribe messaging system. It is a fast, scalable, partitioned and replicated commit log service. Kafka can be used for stream processing, website activity tracking, metrics collection/monitoring and log aggregation applications. The key features of Kafka are scalability, durability, reliability and performance.

The high level architecture of Kafka is shown in Figure 2-3.The primary components of the Kafka are;

- Topic: Stream of messages
- Producer: Anyone who publishes a message to a topic
- Consumer: Anyone who subscribes one or more topics and pull messages.
- Broker: Server in a cluster

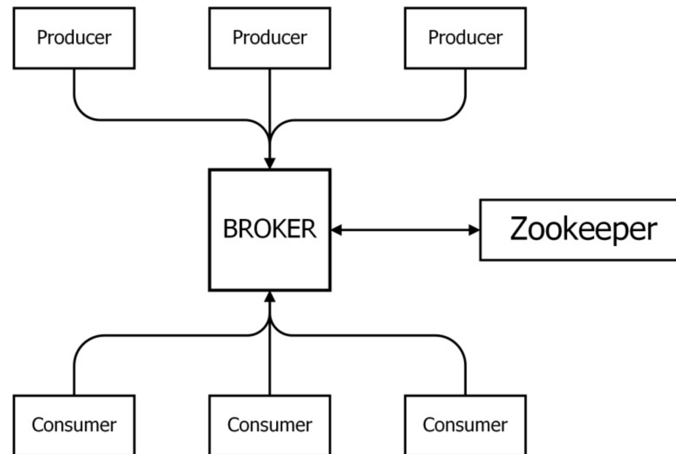


Figure 2-3. High Level Architecture of Kafka

Kafka uses Zookeeper to maintain coordination among nodes. The main differences between the Kafka and the other messaging systems are;

- Kafka is easy to scale out,
- Kafka provides high throughput for both publishers and subscribers,
- Kafka stores messages on disks for batched consumptions.

2.7 Node Red

Node Red is produced for Internet of Things (IoT) solutions by IBM. It provides a visual editor to wire/connect the stream of events and hardware/APIs. The Node Red is based on Node.js [17]. Thus, all of the functionalities are represented as “Node”. Each node has an HTML and a JavaScript implementation. The HTML implementation runs on a web-browser and the JavaScript implementation runs on the server.

In Node Red, users use a web-based visual editor to bring nodes together to create an application flow. The application flows generate JSON files that contain the how nodes are connected to each other and the parameters of each node. The basic application/flow of Node-Red is shown in Figure 2-4.

Figure 2-4. Basic Node Red Flow

2.8 Zookeeper

Zookeeper is an open source service that provides synchronization among clusters for distributed systems. The key features of Zookeeper are speed, reliability, robustness and simplicity. It provides high availability and high throughput with low latency. Kafka, Storm and HBase use Zookeeper to ensure coordination across processes that run on different nodes.

CHAPTER 3

CONTINUOUS QUERIES

Continuous query (CQ) is a persistent query that allows users to receive new results when they become available [21]. Most of the existing studies in the literature propose a SQL like language with SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY and LIMIT clauses to define continuous queries.

In this thesis, we propose a data-flow based query definition model that allow users to define flexible queries and enrich the content of the queries. With the help of data-flow based query model, users are able to define continuous queries for a wide variety of applications. Smart home systems, outlier detections, environment monitoring, M2M communications and smart road applications are some of the use cases of the proposed query model. The proposed query definition model is explained in Section 3.1. In Section 3.2, some of the motivating use cases are listed and described.

3.1 The Proposed Continuous Query Model

The proposed system uses a data flow based query definition model with boxes and arrows. Users design a directed graph of the query with boxes and arrows. The boxes represent well-defined operations and data sources. Each operation takes one or more input streams and produces one or more outputs. The input and output stream directions are indicated by arrows. The basic query model of the proposed system is shown in Figure 3-1.

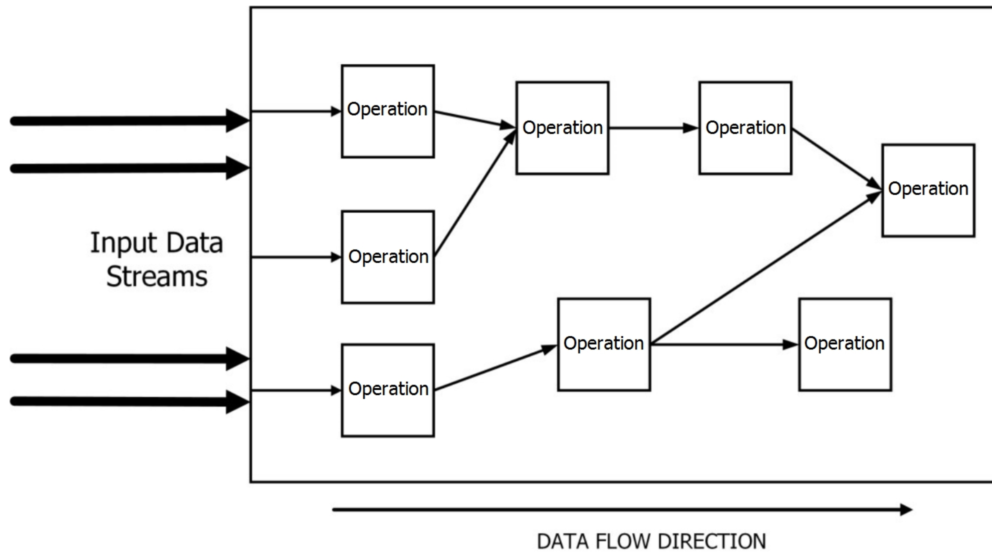


Figure 3-1. The Query Model

In this query model, input data streams flow through directed graph of operation boxes. The boxes may be responsible for listening data source(s), processing the received tuples to produce output, store the received tuples and/or forward them to user via different channels like Twitter or web service.

Users are able to design continuous queries by placing and wiring operation boxes on the Node-Red visual interface. A sample query is shown in Figure 3-2. In this sample query, the "sensor" box listens to the data source(s) and forwards the received tuples to "Calculate Average" box. The "Calculate Average" box calculates the average of the received sensor readings and forwards it to "Store in HBase" box. The "Store in HBase" box stores the received sensor readings in HBase database management system.

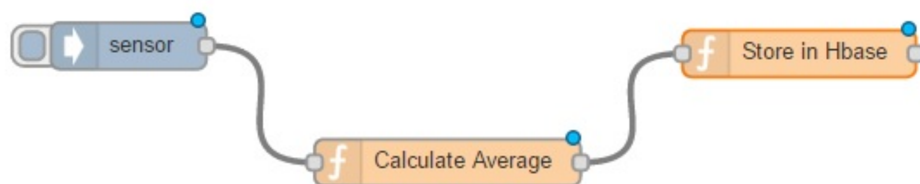


Figure 3-2. Sample Query

3.2 Query Elements

In this thesis, one data source and fourteen generic operations are defined and used. These are:

- Sensor: A data source that is responsible for providing data streams defined in a user query. This data stream is converted to tuples and sent to split sensor data operation.
- Split Sensor Data: Split received tuples into sensor id and sensor reading fields. It is also used to distribute sensor readings to different operations.
- Average: Calculates the average of received readings.
- Moving Average: Calculates the moving average of received readings according to a user defined window size.
- Sum: The sums of the readings are calculated within a defined time window.
- Maximum: Finds the peak point of sensor readings within a defined time window.
- Minimum: Finds the base point of sensor readings within a defined time window.
- Compare Sensors: Compare the latest reading of a sensor with other sensors' latest readings.
- Compare Sensor with Threshold: Compare the readings of a sensor with user defined threshold value.
- Groovy Script: It is responsible for executing Groovy Scripts in run-time.
- Twitter: This bolt sends the received tuples as notification to user via Twitter [40].
- HBase: It is responsible for storing reading tuples into HBase. The file distribution and reliability are handled by Hadoop infrastructure.
- Web Service: This bolt call pre-defined web service for storing reading tuples with user-defined methods.
- Socket: This bolt sends pre-defined message to user defined IP and Port via TCP connection.

3.3 Motivating Examples

The usage area of the proposed query model covers almost all of the IoT application types. In order to demonstrate the applicability of the proposed query model in IoT domain, three different sample applications are designed in this thesis;

1. Traffic Lights Management System for Four Leg Intersection Roads
2. Home Automation System
3. Sport Tracker System

3.3.1 Traffic Lights Management System for Four Leg Intersection Roads

The aim of this query is changing the state of the traffic lights to minimize the wait time of the cars at the intersection and to give priority to ambulances, fire trucks and police cars. The sample sketch of four-leg intersection road is shown in the Figure 3-3.

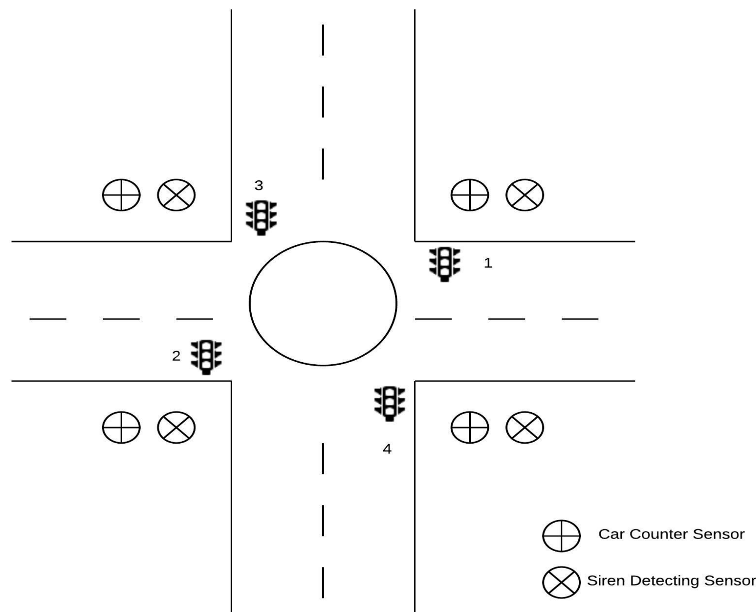


Figure 3-3. The Four-Leg Intersection Road Sketch

The continuous query created for the traffic lights management system for four leg intersection roads is shown in Figure 3-4.

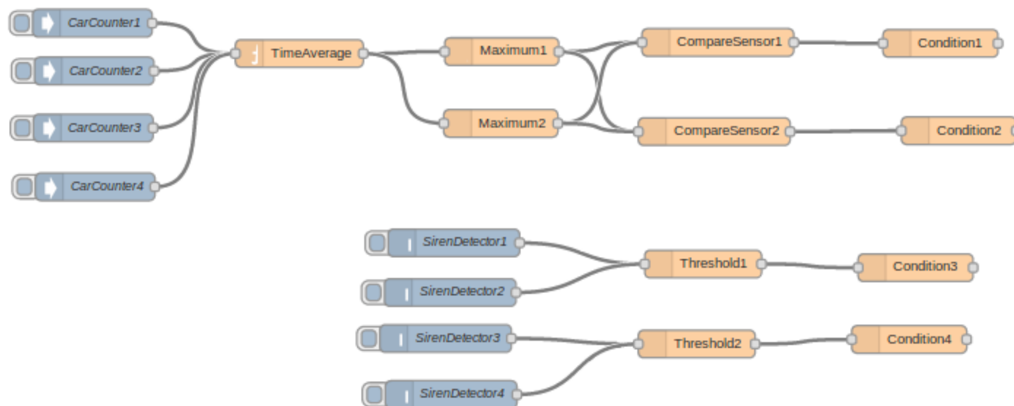


Figure 3-4. Traffic Lights Management System Query

The data sources of this query are car counter and siren detecting sensors. The output of the query is a set of actuator inputs and configuration values which include the duration of the red and green lights for each traffic light in the intersection road. The operation boxes calculate the average of the cars that pass through the each traffic light in every 100 seconds. The calculated averages are compared with each other to determine the state of the traffic lights. This query is also checks the siren detecting

sensors to change the state of the traffic lights. If the system detects the siren, the state of the traffic lights, which detects the last siren, is changed immediately.

3.3.2 Home Automation System

In this scenario, a home automation system is designed to control different parts of a dwelling. The system uses a thermostat, a doorbell, a light detector and a gas detector. The system continuously checks the temperature with upper limit and lower limit to open/close the thermostat. The system checks the gas detector and doorbell once in a second to warn user immediately for an emergency. Moreover, the light sensor checks the level of light to turn on/off the lights. The created query for the Home Automation System is shown in Figure 3-5.

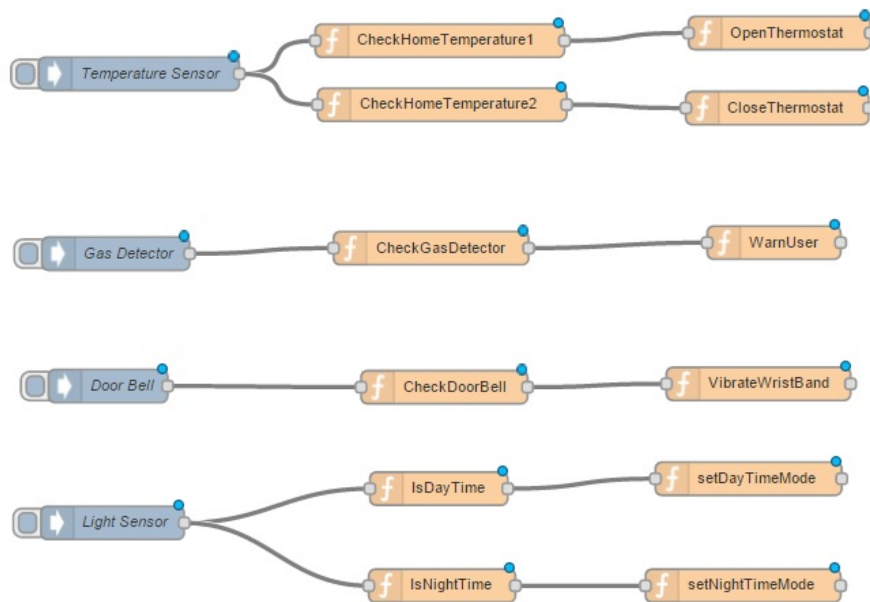


Figure 3-5. Home Automation System Query

3.3.3 Sport Tracker System

In this scenario, the system continuously checks the state of the health of the user with a heart rate sensor and a body temperature sensor. The system warns the user immediately when the heart rate or body temperature exceeds the user defined thresholds. Moreover, the average of heart rate, body temperature and the number of paces are stored in the database to track progression. The created query for the Sport Tracker is given in Figure 3-6.

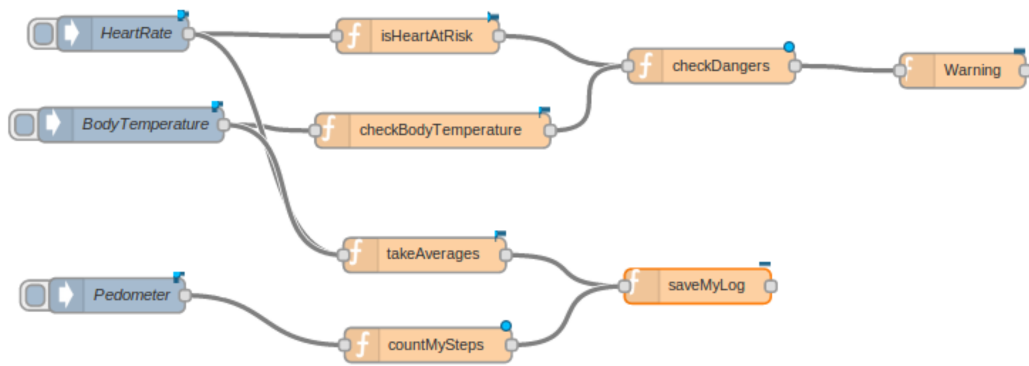


Figure 3-6. Sport Tracker System Query

CHAPTER 4

ARCHITECTURE

The objective of this study is processing large number of continuous queries in an effective manner over data streams and notifying users in real time by providing a generic and a scalable architecture. In the proposed architecture, we applied separation of concerns, low coupling and high cohesion principles for modularity. Thus, any system module can be replaced with another one providing the same functionality. Moreover, the system modules can be improved by adding new functionalities, without affecting the other system modules.

This chapter gives an overview of the framework and its architecture. In Section 4.1, we provide the detailed explanation of the proposed architecture. Section 4.2 explains how users can implement and run a continuous query by utilizing the proposed framework.

4.1 System Architecture

In this section, we describe how queries are generated and represented, how these queries are processed in real time, how sensor data is distributed to Storm topologies and finally how these modules are integrated to process a multitude of continuous queries in real time.

We have utilized the state-of-the-art cloud and big data technologies, which are Storm, Kafka, HBase, Zookeeper and Node-Red in our architecture to propose a solution for the continuous query execution problem. In the proposed architecture, we have utilized Kafka, HBase and Zookeeper as they are. Storm and Node-Red are modified according to our needs. Node-Red is used as the visual editor to design continuous queries but those queries are not run on the Node-Red server.

We have also changed the way of using Storm. The Storm is designed to automatically build the network of queues and workers to do real-time processing [3]. However, we have utilized Storm only for running continuous queries over distributed JVMs.

The system architecture is delineated in the Figure 4-1. The system consists of the following main components:

- Query Generation: This module enables users to create continuous queries. Creating a query requires forming a graph by using basic query elements and relating sensors/actuators to those elements. The system users can design a query by utilizing the drag and drop visual interface of Node Red. There is an additional Graphical User Interface (GUI) component that is used in the editor for searching sensor ids, specifying query duration and forwarding query to Cloud Application Server (CAS).
- Query Processing: The user defined continuous queries are processed with query specific Storm topologies and each topology represents a query-processing unit.
- Data Distribution: Listens to all of the data sources and distributes sensor readings to proper processing units. Kafka publish-subscribe system is used for this purpose.
- Cloud Application Server (CAS): Provides synchronization among system modules, gets queries from users, stores meta-data of queries/users/sensors in a relational database and controls the execution of the query processing units.
- Sensors: Detects/measures the input for the system from the physical environment.
- Relational Database: It is a set of pre-defined tables to store and access the sensor(s) and user information.

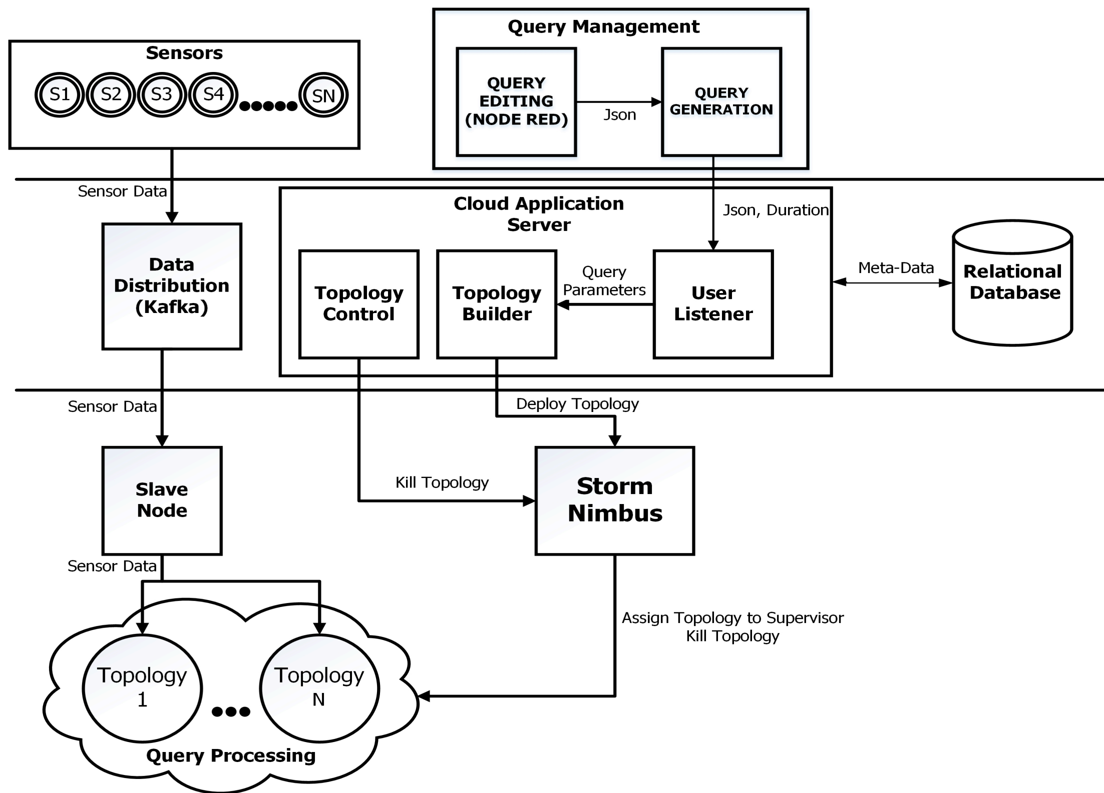


Figure 4-1. System Architecture

In this system, users can define continuous queries to process, analyze, react to, report and monitor a large amount of sensor data in real time. The continuous queries may consist of some statistical computations, rules, event/time based triggers, Groovy scripts, notifications and/or data storage. The created queries are sent to CAS by the user. The CAS builds a query specific Storm topology and stores the meta-data of the query in a relational database to generate a unique key. This unique key is used to control the life cycle of the topology. The topologies are deployed in a distributed computing environment with the help of Storm. Each topology subscribes to a Kafka topic to receive relevant sensor data, which is defined in the query. The Data Distribution module listens to all data sources and publishes received sensor data in the proper Kafka topic. Users also specify how to get notifications from system in response to queries. The notifications can be sent to the user as soon as a rule/condition is satisfied and/or when the query processing is over.

4.1.1 Query Generation and Representation

There are four steps to define a query: specifying sensor ids, defining the query as a data flow graph with the Node Red visual interface, defining a query duration to specify how long the query will run in seconds and sending the query to CAS to deploy query. The queries are defined as a directed data flow graph with the Node Red visual interface. The searching sensor(s), query duration definition and forwarding query to CAS are handled by an additional GUI component.

Users define a data source by selecting at least one sensor. The sensors are presented in a hierarchical structure according to facilitate sensor selection. There are four levels in the hierarchy:

1) Location Level: Locations, for instance a city or a university.
2) Place Level: There may be several places in a specific location. For example, a department in a university or a neighborhood of a city.
3) Node Level: Gateways serving a section in a place are referred to as Nodes. For instance, nodes can serve sensors in a room, in a building or in a street (or part of a street).
4) Sensor Level: One or more sensors can be directly or indirectly attached to a node. Each sensor is connected to the Internet via one node. System also keeps detailed information about the type and range of measurements for each sensor.

The query functionalities/operations are implemented as a “box” in Node-Red. Each box represents an operation that will be performed on the received data. One or more data sources must be specified for the boxes at first, and then the boxes are wired to compose the logic of the query. The list of operations/boxes is given in Section 3.2 and the implementation of the queries for each operation/boxes is described and illustrated in Section 4.2.

The Node Red generates a JSON file, which contains how operation boxes are connected to each other and the parameters of each box. The user uploads the JSON file of the query and specifies how long the query will run be in seconds. The JSON file is parsed by Query Generation module. Query is represented as a directed graph data structure. This directed graph structure and the duration of the query are sent to CAS in a data packet through a TCP connection.

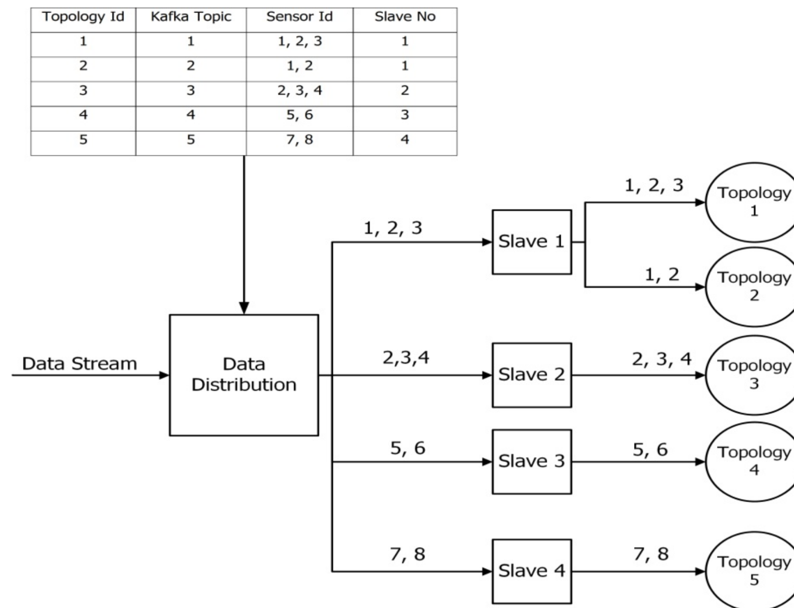
4.1.2 Query Processing

The query-processing module is based on Storm. In our architecture, a query corresponds to a Storm topology. The topologies are formed according to the directed data flow graph of the queries. Each Storm spout represents a sensor node (i.e., data source) in the query and each Storm bolt represents an operation box in the query. Storm spouts and bolts are designed modularly and they emit data in the same format. Hence bolts can be interconnected in different ways. The defined spout/bolts cover the needs of most of the IoT application scenarios. The proposed architecture is also open for improvements with implementation of new bolts as well.

4.1.3 Data Distribution

Each topology subscribes to a Kafka topic to receive relevant data from one or more sensors that are defined in the query. The CAS sends the received sensor readings to proper slave nodes. The slave nodes distribute the received sensor readings from master node to running topologies. Each topology subscribes to a Kafka topic(s) to receive relevant data from one or more sensors that are defined in the query as the

data source(s). In order to achieve this, the sensor ids' are related to Kafka topics. The data distribution is illustrated in Figure 4-2.



Topology Id	Kafka Topic	Sensor Id	Slave No
1	1	1, 2, 3	1
2	2	1, 2	1
3	3	2, 3, 4	2
4	4	5, 6	3
5	5	7, 8	4

Figure 4-2. Data Distribution Module

Each slave node runs a local server, which communicates with master node to subscribe/unsubscribe to a topic. The topologies send a TCP message, which contains the list of sensor ids and topology id to their local server before starting execution. According to received message, the local server request a subscription/un-subscription to a topic from master node. In other words, the master node manages distributing data to slave nodes and a slave node is responsible for distributing sensor readings to the topologies that runs on that slave node. The sensor readings are sent to each slave node at most once with this approach.

4.1.4 Cloud Application Server

The Cloud Application Server (CAS) is responsible for getting queries from users, storing meta-data of sensors/queries/users and controlling the execution of the topologies.

The generated query flows are sent to CAS in a data packet through a TCP connection. The data packet contains a query object as a directed graph form and the duration of the query in seconds. The query object specifies how bolts are interconnected to each other and the relevant parameters of operations. The query names are stored in the relational database. The primary key of a query serves as the topology identifier and also the Kafka topic name of the topology. Hence, the Topology Builder unit builds a query specific topology according to the parameters inside the query object. The deployed topology is assigned to a supervisor component of a slave node. Each bolt and spout works on exactly a single thread and each topology works on exactly one worker. Distributing topologies and assigning jobs to a supervisor is under the responsibility of Storm.

The end time of each query is calculated when the query is deployed to Storm, and the calculated end time is stored in the memory. A timer task thread is responsible to control the life cycle of each topology. With the timeout, the topology is killed. In order to deploy or to kill a topology, Topology Control unit in CAS communicates with the Storm Nimbus client. The Nimbus client communicates with proper supervisor to kill the topology. The Nimbus client also keeps the list of running topologies and the meta-data of the topologies.

4.2 Query Implementation

User composes a query by wiring a set of nodes in Node-Red visual interface. The composed query is transformed to a topology by the Storm system. As all of the nodes are designed modularly with a well-defined uniform interface, nodes can be interconnected in different ways. The nodes (except HBase) take two inputs; sensor id and sensor data. In addition to these two, the HBase node takes the name of the node as an input to generate database row key. The nodes emit the output; sensor id and the processed data. The description of nodes, parameters and example usage of each node are explained in the following sections.

4.2.1 Sensor

This node is used for specifying data source(s). For each sensor node, user assigns at least one sensor id. There must be a sensor node in all queries because the sensor id(s) defined in sensor node(s) is used to create a topology specific Kafka topic.

4.2.2 Average

This node is responsible for calculating the average of received sensor readings. It takes three parameters: ID, Time and Emit. The description of the parameters and an example query is shown in Table 4.1.

Table 4.1. Average Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	The sensor id(s) for which the average(s) will be calculated.
Time	NO	Infinite	The time interval (in seconds) to reset the calculated averages. By default, the calculated averages will not be reset.
Emit	NO	Sensor id(s)	The sensor id(s) for which the average(s) is emitted to other bolts.
Example Query	ID, SensorID1 Time, 100 Emit, SensorID1		

4.2.3 Sum

This node is responsible for keeping track of the number of the readings and sum of readings. It takes three parameters: ID, Time and Emit. The description of the parameters and an example query is shown in the Table 4.2.

Table 4.2. Sum Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	Specifies the sensor id(s) for which the reading sum(s) will be calculated.
Time	NO	Infinite	Determines time interval in seconds to reset the calculated summations. By default, the calculated sum(s) will not be reset.
Emit	NO	Sensor id(s)	Specifies the sensor id(s) for which the sum emitted to other bolts.
Example Query	ID, SensorID1 Time, 100 Emit, SensorID1		

4.2.4 MovingAverage

MovingAverage node is responsible for calculating the moving average of the received sensor readings. There are four parameters: ID, Time, Window Size and Emit. The description of the parameters and an example query is shown in the Table 4.3.

Table 4.3. Moving Average Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	Specifies the sensor id(s) for which the average(s) will be calculated.
WindowSize	YES	-	The number readings, N, used to calculate the average.
Time	NO	Infinite	Determines time interval in seconds to reset the calculated moving averages. By default, the calculated moving average(s) will not be reset.
Emit	NO	Sensor id(s)	Specifies the sensor id(s) for which the average(s) is emitted to other bolts
Example Query	ID, SensorID1 WindowSize, 5 Time, 100 Emit, SensorID1		

4.2.5 Max

This node finds the largest reading for each input sensor. It takes three parameters: ID, Time and Emit. The description of the parameters and an example query is shown in the Table 4.4.

Table 4.4. Max Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	Specifies the sensor id(s) for which the maximum reading will be found.
Time	NO	Infinite	Determines time interval in seconds to reset the found peak point(s). By default, the determined peak points will not be reset.
Emit	NO	Sensor id(s)	Specifies the sensor id(s) for which the maximum value(s) is emitted to other bolts.
Example Query	ID, SensorID1 Time, 100 Emit, SensorID1		

4.2.6 Min

This node finds the smallest reading for each input sensor. It takes three parameters: ID, Time and Emit. The description of the parameters and an example query is shown in the Table 4.5.

Table 4.5. Min Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	Specifies the sensor id(s) for which the minimum reading will be found.
Time	NO	Infinite	Determines time interval in seconds to reset the found base point(s). By default, the determined base point will not be reset.
Emit	NO	Sensor id(s)	Specifies the sensor id(s), which are emitted to other bolts.
Example Query	ID, SensorID1 Time, 100 Emit, SensorID1		

4.2.7 CompareSensors

This node is responsible for comparing the latest sensor readings. There are four parameters: ID, Mask, Compare and Emit. The description of the parameters and an example query is shown in the Table 4.6.

Table 4.6. CompareSenssors Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	Specifies the sensor id(s) for which the readings will be compared.
MASK	NO	AND	The masking operation that will be applied for compare operations. The valid values are “AND”, “OR”.
Compare	YES	-	Specifies the sentence that defines the compare operation.
Operation	YES	-	The comparison operator that will be applied to sensor readings, the valid operators are:>,<,>=,<=
Emit	NO	Sensor id(s)	Specifies the sensor id(s) for which the result(s) is emitted to other bolts.
Example Query	MASK, OR Compare, ID, SensorID1, Operation, >, ID, SensorID2 Compare, ID, SensorID2, Operation, <, ID, SensorID3		
Mathematical Representation	(SensorID1>SensorID2) OR (SensorID2<SensorID3)		

4.2.8 Threshold

This node is responsible for comparing the latest sensor readings with a threshold. There are three parameters: Mask, Compare and Emit. The description of the parameters and an example query is shown in the Table 4.7.

Table 4.7. Threshold Node Parameters

Parameter Name	MUST	Default Value	Description
ID	NO	Sensor id(s)	Specifies the sensor id(s) to compare readings with the user-defined threshold.
MASK	NO	AND	Indicates which masking operation will be applied for compare operation.
Compare	YES	-	Specifies the sentence that defines the compare operation.
Operation	YES	-	Indicates which comparison operator will be applied for readings, >,<,>=,<=.
Threshold	YES	-	Specifies the threshold.
Emit	NO	Sensor id(s)	Specifies the sensor id(s) for which the result(s) is emitted to other bolts.
Example Query	MASK, OR Compare, ID, SensorID1, Operation, >, Threshold, N Compare, ID, SensorID2, Operation, <, Threshold, N Compare, ID, SensorID3, Operation, >, Threshold, N		
Mathematical Representation	(SensorID1>N) OR (SensorID2<N) OR (SensorID3>N)		

4.2.9 Groovy Script

It is possible to execute groovy scripts during run-time. Users can implement their own groovy scripts to execute on streaming data. We provide a Java object to facilitate script implementation. The class diagram of this Java object is given in the Figure 4.3.

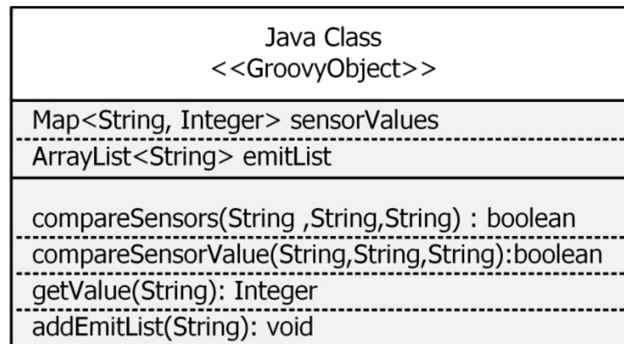


Figure 4-3. Query Object Class Diagram

The "sensorvalues" map object contains the list of sensor id(s) and the current values of each sensor(s). The "emitList" contains the sensor id(s) for which the result(s) will be emitted to the next bolt. User can compare sensor readings with another sensor

reading with the help of “compareSensors” method. This method takes three input parameters which are sensor id for the first sensor, the comparison operator such as “<, >, <= or >=” and sensor id for the second sensor. Moreover, user may set a threshold for a sensor reading with “compareSensorValue” method. This method also takes three input parameters; sensor id for the sensor, the operation signature and the threshold.

An example groovy script to compare sensor with another sensor reading and with threshold to add emit list is given in Table 4.8.

Table 4.8. Example Groovy Script

Example Query Script	if(groovyObject.compareSensors(Sensor1,">",Sensor2)
	&&groovyObject.compareSensorValues(Sensor”,”<”,10)){
	addEmitList(“1”); }

4.2.10 Twitter

This node sends the received tuples as notifications to the user via Twitter [40]. There is only one parameter, which is username to indicate the twitter username of the user. The description of the parameters and the example query is shown in the Table 4.9.

Table 4.9. Twitter Node Parameters

Parameter Name	MUST	Default Value	Description
Username	YES	-	Indicates the twitter username.
Example Query	@twitterUserName		

4.2.11 HBase

It is responsible for storing reading tuples into HBase. There is no parameter for this node. The HBase bolt stores the tuples according to received bolt name.

4.2.12 Web Service

This node is responsible for calling pre-defined web service for storing reading tuples with user-defined methods. There is only one parameter, URL that specifies the web service URL. The description of the parameters and an example query is shown in Table 4.10.

Table 4.10. Web Service Node Parameters

Parameter Name	MUST	Default Value	Description
URL	YES	-	Indicates the web service URL.
Example Query	URL, http://localhost:9999/ws/hello?wsdl		

4.2.13 Socket

This bolt sends a pre-defined message to a user defined host specified by an IP address and a Port number via TCP connection. There are three parameters, IP, Port and Message. The description of the parameters and an example query is shown in Table 4.11.

Table 4.11. Socket Node Parameters

Parameter Name	MUST	Default Value	Description
IP	YES	-	Specifies the IP address of the server.
Port	YES	-	Specifies the Port number used by the server.
Message	YES	-	Specifies the message to send.
Example Query	IP, 192.168.1.21 Port, 10008 Message, Set Condition		

CHAPTER 5

PROTOTYPE IMPLEMENTATION

In Chapters 3 and Chapter 4, the objectives, the sample use cases and the system architecture of the proposed framework are described. As explained in Chapter 4, the system consist of four primary modules: Query Generation, Data Distribution, Query Processing and Cloud Application Server. This chapter explains the implementation details of the prototype.

5.1 Query Generation Module

The Query Generation module is responsible for assisting users in defining queries and sending the queries to CAS. This module is also responsible for converting the query to directed graph object that can be executed in the Storm environment. There are 5 classes in this module; GUI, ParseQuery, DataPacket, QueryFlow and Node (Figure 5.1).

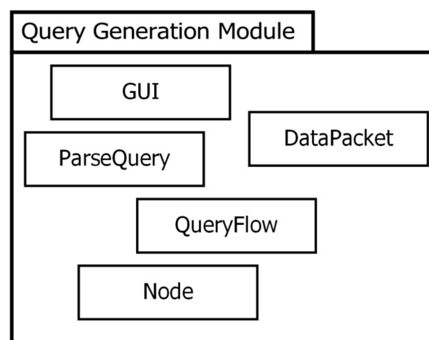


Figure 5-1. Query Generation Module Package Diagram

- GUI: It contains the GUI implementation that enables users to search sensor id(s) and send query to CAS. This class reads the sensor information from the relational database. The sample screenshots of the GUI can be seen in the Figure 5.2.
- Parse Query: The user-defined queries are stored as a JSON file created by Node Red. The user uploads this JSON file to the system via the GUI. The uploaded file is then parsed to represent query as a directed graph object in Java environment.
- Query Flow: This class represents the directed graph structure. It contains the nodes which represent the operations and the edges which represent the directions of the data flow.
- Node: It is the vertex of the graph structure and it contains the input/output links, name, type and the parameters of the node.
- Data Packet: This class is used to forward query to CAS. It contains, package header, selected sensor ids', query duration and the query flow object.

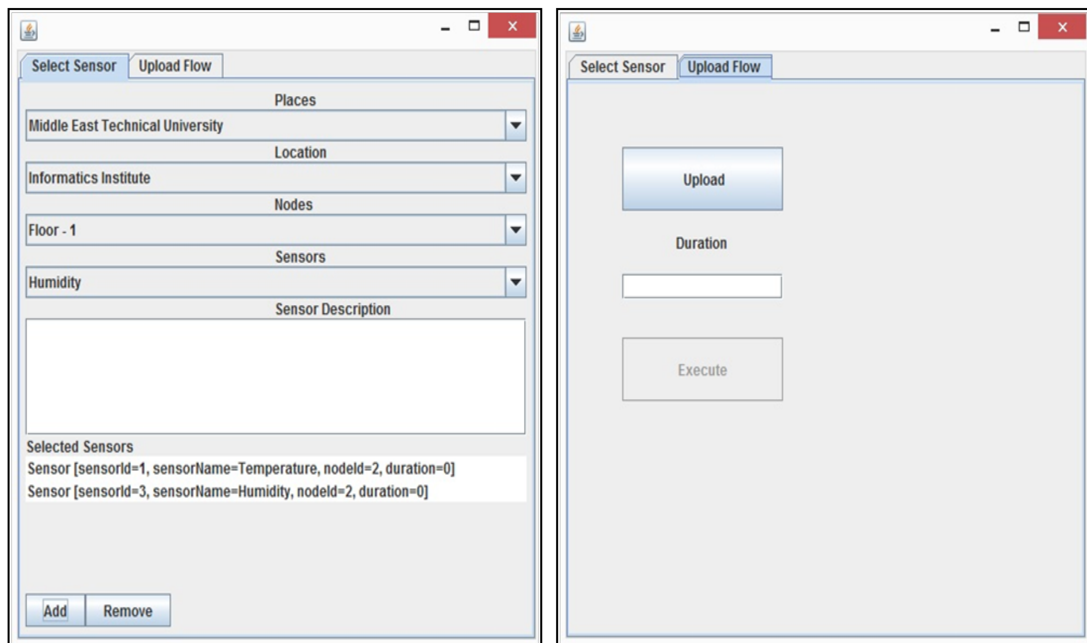


Figure 5-2. GUI Screenshots

An HTML fragment is created for each node/operation type. The HTML fragment contains the visual details; labels, text-boxes, colors and etc. that serve as the visual interface of the nodes in the Node Red environment. The sample screenshot of the Node Red visual interface that displays a node's details is shown in the Figure 5-3.

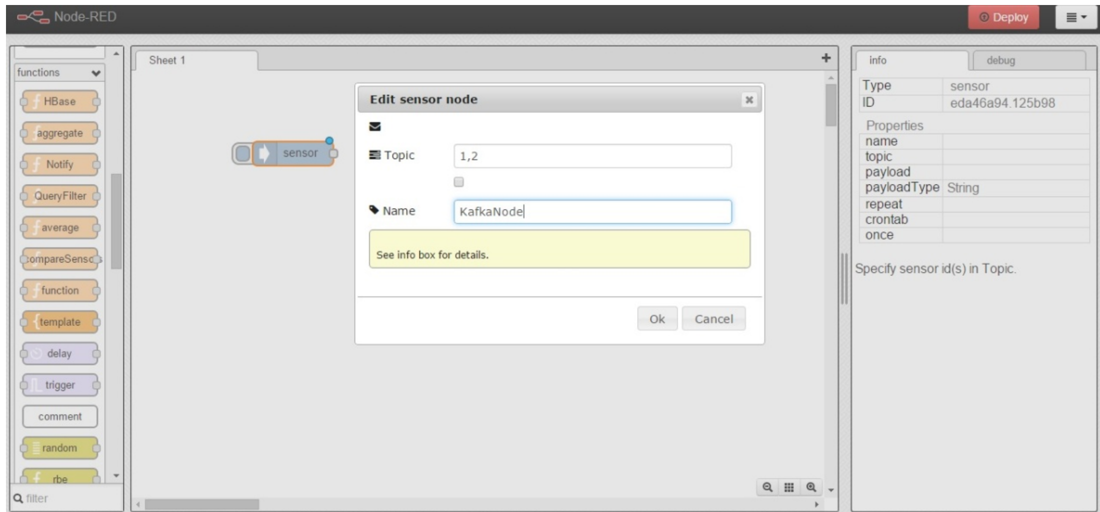


Figure 5-3. Node Red Visual Interface

5.2 Data Distribution Module

The Data Distribution module is based on Kafka. We have also implemented a local server for internal data distribution. The package diagram of the Query Generation module is shown below.

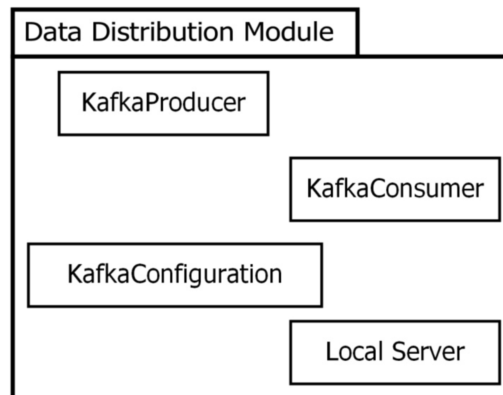


Figure 5-4. Data Distribution Module Package Diagram

- Kafka Producer: Ordinarily, the data is collected from the sensors distributed on the field. However, for practical reasons, we have simulated a set of sensors for the performance evaluation. This class is responsible for producing random sensor readings and publishing in the "sensorData" Kafka topic. The produced sensor readings consist of two integers; sensor id and sensor data. The sensor readings are produced at random time intervals. The time intervals are identified with exponential distribution function to simulate network latency. The data is delivered to CAS with default Kafka broker port which is "9092".

- Kafka Consumer: It is responsible for distributing sensor readings to proper slave nodes. Thus, it keeps the name of the slave nodes and the sensor id(s)

that are subscribed by the slave node in a hash-map data structure. The key is the slave node name and the value is the list of sensor id(s). The hash-map is updated when a slave node subscribes/unsubscribes to a topic. According to the hash-map content, the sensor readings received from Kafka broker are published in the proper Kafka topic, which is the name of slave nodes.

- Kafka Configuration: It keeps the Kafka configuration information; the port number, the Zookeeper port number, the list of Kafka brokers and the session time-out duration.

- Local Server: Each slave node runs a local server to listen subscribe /unsubscribe Kafka topic requests of topologies. Thus, the slave nodes also keep the list of topology id(s) and the sensor id(s) that are subscribed by the running topologies in the slave node, in a hash-map data structure where the key is topology id and the value is the list of sensor id. The received sensor readings are published in the proper Kafka topics, which are the topology ids. The spouts of the topologies listen to the corresponding Kafka topic to receive sensor readings.

5.3 Query Processing Module

The Query Processing module is based on Storm. Thus, this module includes the implementation of bolts/spouts and HBase connections. The package diagram of the Query Processing module is shown in Figure 5-5.

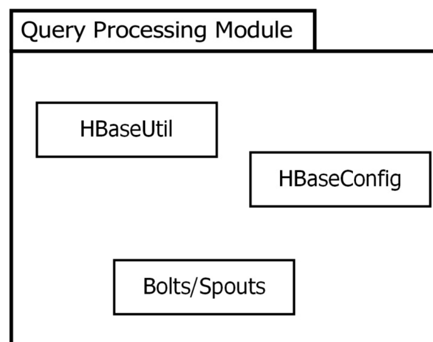


Figure 5-5. Query Processing Module Package Diagram

- Bolts/Spout: Each box/operation/node in a query is implemented in Java Platform as a Storm Bolt/Spout object. The Storm bolt objects contain the methods given in Table 5.1.

Table 5.1. Storm Bolt Methods

Method Name	**Parameters**	**Description**
DeclareOutputFields	OutputFieldsDeclarer declarer	Declares the direction of the output for this bolt.
Prepare	Map, TopologyContext, OuputCollector	This method is called before the bolt starts processing tuples. Thus, it is generally used for initializing variables.
Execute	Tuple	This method contains the logic of bolts. It processes tuples to produce the output.
Cleanup	-	It is called when a bolt is going to shut down. However, there is no guarantee to call this function in the distributed mode.

- HBase Utils: This class contains the methods to insert data to the database, read data from the database and create a table. HBase tables store data as a multidimensional-sorted map, which is indexed by row-keys. In order to find a specific data with HBase API, the system needs to know the row-key for the data. Otherwise the system needs to scan all of the tables and rows to find the required data. Therefore, the row key design determines how the system will communicate with the HBase. The row-key design determines the performance in scanning operation. Each bolt produces a different kind of output in order to identify, which data is stored in database and eight different row-keys are designed to store the output of the topologies. Sample row key designs are given in Table 5.2

Table 5.2. HBase Row Key Designs

Bolt Name	Row Key Design
Compare Sensors	CSensors_TopologyId_SensorId_TimeStamp
Compare Sensor with Threshold	CThreshold_TopologyId_SensorId_TimeStamp
Groovy Script	Groovy_TopologyId_SensorId_TimeStamp
Maximum	MAX_TopologyId_SensorId_TimeStamp
Minimum	MIN_TopologyId_SensorId_TimeStamp
Average	AVG_TopologyId_SensorId_TimeStamp
Sum	Sum_TopologyId_SensorId_TimeStamp
Moving Average	MovingAverage_TopologyId_SensorId_TimeStamp

- HBase Config: It contains the configuration of the HBase to connect Hadoop HDFS. The configuration contains; the IP address of the master Hadoop node, the Zookeeper and the port number of the Zookeeper. In Hadoop setup, the location and the port of the HDFS are defined in the Hadoop configuration file.

5.4 Cloud Application Server Module

The Cloud Application Server (CAS) is responsible for listening to user requests, storing meta-data of sensors/queries/users, and controlling the execution of the topologies. The package diagram of CAS is shown in Figure 5-6.

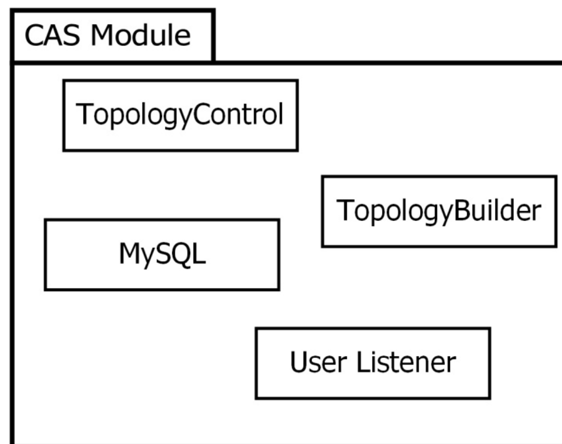


Figure 5-6. Cloud Application Server Package Diagram

- User Listener: This class is responsible for listening requests from users and slave nodes. The requests are delivered to the CAS via TCP connection to the port number "10018". The CAS also listens to the Kafka topic subscribe/unsubscribe requests of Slave nodes. The slave nodes send their requests with a data packet through a TCP connection. The CAS forwards the request to Kafka consumer with a method call. The received requests are stored in the relational database.

- Topology Control: The duration of the query is stored in the memory to control the life cycle of its execution. The end time of each query is calculated when the query is deployed to Storm. A timer task thread is assigned to control the life cycle of each topology. With the timeout, the topology is killed. In order to deploy or to kill a topology, CAS communicates with the Storm Nimbus client. The Nimbus client also keeps the list of running topologies and the meta-data of the topologies.

- MySQL: This class contains the methods to interact with the relational database. We used MySQL as the relational database in our prototype implementation. The meta-data of the created queries are stored in MySQL. The entity-relationship diagram for meta-data is given in the Figure 5.7.

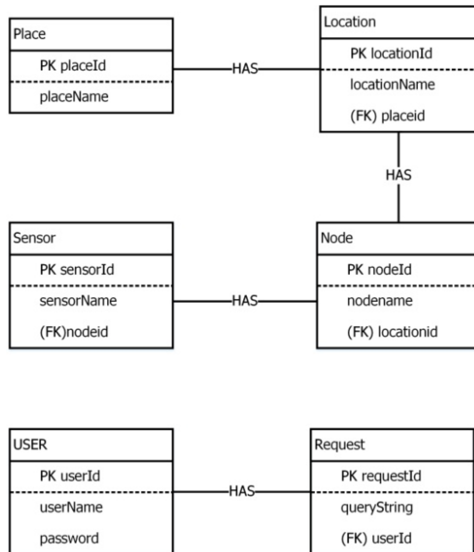


Figure 5-7. Entity Relationship Diagram

- Topology Builder: The Topology Builder converts the received directed graph to a Storm topology, sets the topology configurations and connects Storm Nimbus to deploy the topology.

5.4.1 Deploying a Topology

The topologies are deployed to distributed environment with StormSubmitter. The StormSubmitter takes 4 input parameters: a topology, the name of the topology, the topology configuration and the packaged jar file of the spout/bolts classes. In Storm, each bolt/spout has to know the bolts to receive input and emit output. The topology builder is responsible for building the topology and wiring the spout/bolts according to the directed graph representation of the query. The only difference between the user defined query graph and the Storm topology graph representation is that the Kafka Spout is defined by Topology Builder according to the user defined sensor ids'. A Sensor node in a query is defined as a "Split Sensor Data Bolt" because, users can define more than one data source and defining each data source as Kafka Spout increases the workload of the Kafka broker. Therefore, we define one Kafka Spout for each topology and the user defined data sources are used to split received data and distribute to proper topologies. A sample query and the representation as a Storm topology are shown in Figure 5.8 and Figure 5.9, respectively. In topology configurations, we have defined one worker and one supervisor for each topology. Moreover, single thread is assigned to each bolt and spout. The built topology is deployed on the Storm environment through the Storm Nimbus component. We have created a packaged jar file for our spouts/bolts with Maven Assembly Plug-in [41]. The jar file contains the source code of the spouts/bolts and all the dependencies of the code. This jar file is added to the class path on the worker nodes.

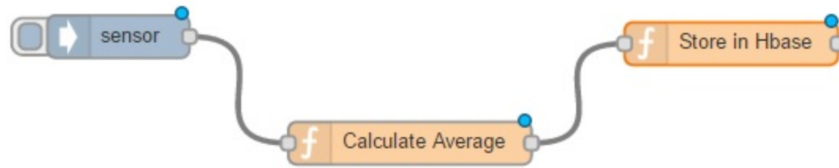


Figure 5-8. Sample Query

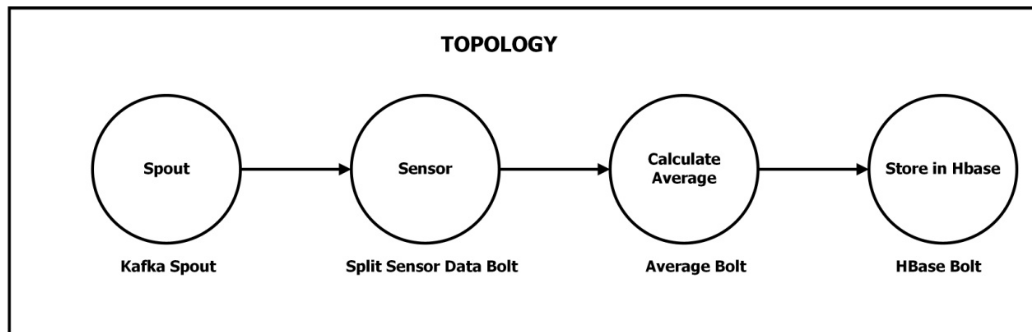


Figure 5-9. Storm Topology Graph of Sample Query

CHAPTER 6

EVALUATIONS

In this chapter, the evaluation of the proposed architecture under different workloads and performance comparison of alternative data distribution approaches are given. The aim of these evaluations is to demonstrate the scalability of the proposed architecture. In accordance with these objectives, the experimental setup is explained in Section 6.1. The test scenarios that are used in evaluations are described and illustrated in Section 6.2. In Section 6.3, the scalability of the proposed architecture is evaluated on two different configurations; one slave node and four slave nodes. The comparison of alternative data distribution approaches are evaluated in Section 6.4.

6.1 Experimental Setup

In order to evaluate the proposed architecture, a distributed system that consists of a master node and four slave nodes is formed. The master node is responsible for assigning and managing jobs of slave nodes. Slave nodes are worker nodes that are responsible for performing specific functions and handling queries. The system architecture embodies two different distributed ecosystems: Hadoop (version 2.6.0) and Storm (version 0.9.2). The deployment diagram of our system which depicts the nodes and the software running on them is shown in Figure 6.1.

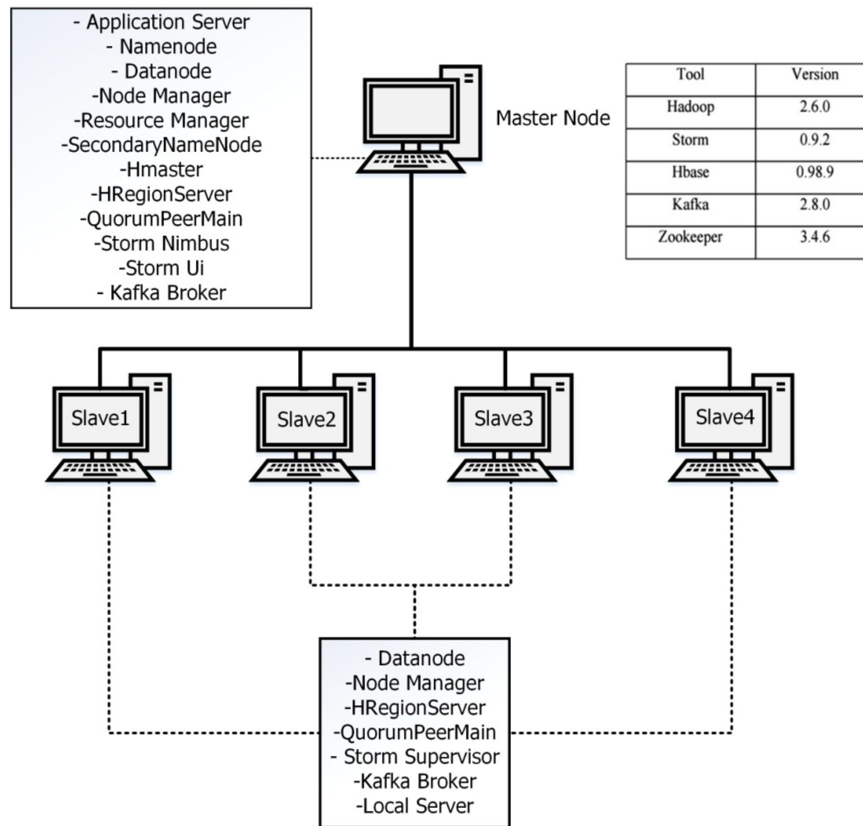
The Master and Slave nodes are created as virtual machines defined on XenServer [36] server virtualization platform. These virtual servers have the same processor configuration: Intel® Xenon® CPU E5-26500, 2 GHz. The master node has 8GB memory and each of the slave nodes has 4GB memory. All of the machines run Ubuntu 14.04 LTS operating system.

In the experimental setup, a sensor simulator produces some random sensor readings. The generated sensor data consists of two integer values: sensor id and sensor reading. The sensor readings are produced at random time intervals. The time intervals are generated according to the exponential distribution function, which is;

$$x = \lambda e^{-\lambda x} \text{ (Equation 6.1)}$$

where λ is the rate parameter.

The generated data is published with “sensorData” topic by Kafka.



Tool	Version
Hadoop	2.6.0
Storm	0.9.2
Hbase	0.98.9
Kafka	2.8.0
Zookeeper	3.4.6

Figure 6-1. System Deployment Diagram

6.2 Scenarios

By using the experimental setup, the scalability of the proposed architecture is investigated on three different scenarios:

1. Traffic Lights Management System for Four Leg Intersection Roads
2. Home Automation System
3. Sport Tracker System

The details of these scenarios can be found in Chapter 3. The implementations of the pertinent queries are explained in this section.

6.2.1 Traffic Lights Management System for Four Leg Intersection Roads

The defined query flow for this scenario is shown in Figure 6.2.

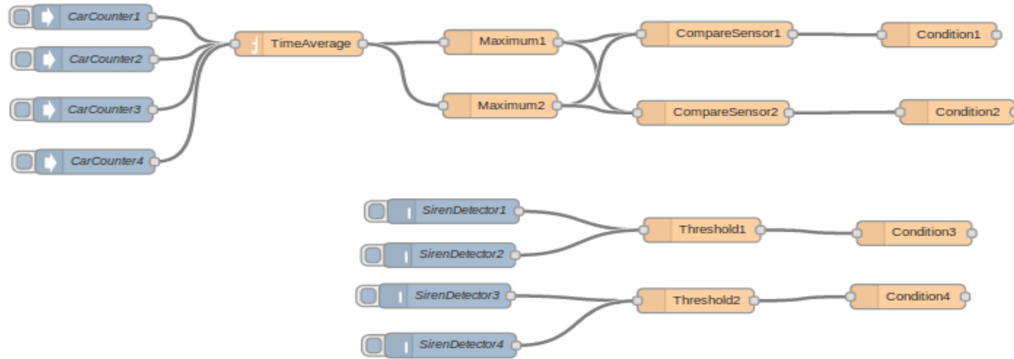


Figure 6-2. Traffic Lights Management System for Four-Leg Intersection Roads

In this scenario, the sensors with ids' equal to 1,2,3,4, are car counter sensors and the siren detecting sensors' ids' are 4, 5, 6 and 7. The implemented queries for each node are given in Table 6.1.

Table 6.1. Queries for Scenario 1

Node Name	Query	Node Type
Car Counter 1	Topic, 1	Sensor
Car Counter 2	Topic, 2	Sensor
Car Counter 3	Topic, 3	Sensor
Car Counter 4	Topic, 4	Sensor
Siren Detector 1	Topic, 5	Sensor
Siren Detector 2	Topic, 6	Sensor
Siren Detector 3	Topic, 7	Sensor
Siren Detector 4	Topic, 8	Sensor
Time Average	Time, 100	Average
Maximum 1	Time, 100	Max
Maximum 2	Time, 100	Min
Compare Sensors 1	MASK, OR Compare, ID, 1, Operation, >, ID, 3 Compare, ID, 1, Operation, >, ID, 4 Compare, ID, 2, Operation, >, ID, 3 Compare, ID, 2, Operation, >, ID, 4	Compare Sensors
Compare Sensor 2	MASK, OR	Compare Sensors

	Compare, ID, 3, Operation, >, ID, 1 Compare, ID, 3, Operation, >, ID, 2 Compare, ID, 4, Operation, >, ID, 1 Compare, ID, 4, Operation, >, ID, 2	
Threshold 1	Mask,OR Compare, ID, 5, Operation, >, Threshold, 2 Compare, ID, 6, Operation, >, Threshold, 2	Threshold
Threshold 2	Mask,OR Compare, ID, 7, Operation, >, Threshold, 2 Compare, ID, 8, Operation, >, Threshold, 2	Threshold
Condition 1, 2, 3, 4	-	Log

6.2.2 Home Automation System

The defined query flow for this scenario is shown in Figure 6.3.

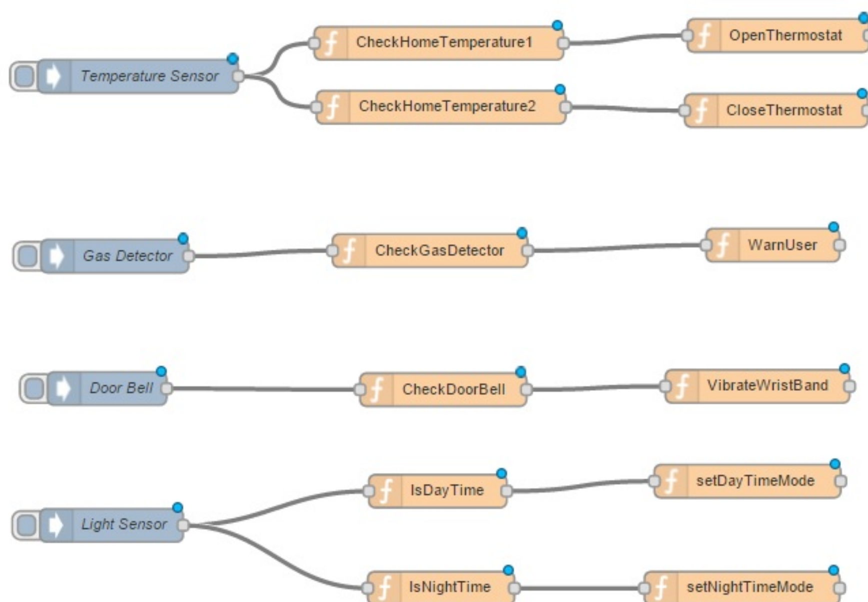


Figure 6-3. Home Automation System

In this scenario, there are four different sensors: a temperature sensor, a gas detecting sensor, a doorbell and a light sensor. The temperature sensor's id is 1, gas detecting

sensor’s id is 2, and the doorbell has the id number 3 and the id is 4 for the light sensor. The implemented queries for each node are specified in Table 6.2.

Table 6.2. Queries for Scenario 2

Node Name	Query	Node Type
Temperature Sensor	Topic, 1	Sensor
Gas Detector	Topic, 2	Sensor
Door Bell	Topic, 3	Sensor
Light Sensor	Topic, 4	Sensor
Check Home Temperature 1	Compare, ID, 1, Operation, <, Threshold, 22	Threshold
Check Home Temperature 2	Compare, ID, 1, Operation, >, Threshold, 25	Threshold
Check Gas	Compare, ID, 2, Operation, >, Threshold, 2	Threshold
Check Door Bell	Compare, ID, 3, Operation, >, Threshold, 2	Threshold
Is Day Time	Compare, ID, 4, Operation, >, Threshold, 5	Threshold
Is Night Time	Compare, ID, 4, Operation, <, Threshold, 3	Threshold
Open Thermostat	IP, localhost Port, 10018 Message, Open Thermostat	Socket
Close Thermostat	IP, localhost Port, 10018 Message, Close Thermostat	Socket
Warn User	@sensorquerytest	Twitter Notify
Vibrate Wrist Band	IP, localhost Port, 10018 Message, Vibrate Wrist Band	Socket
Set Day Time Mode	IP, localhost Port, 10018 Message, Day Time	Socket
Set Night Time Mode	IP, localhost Port, 10018 Message, Night Time	Socket

6.2.3 Sport Tracker System

The defined query flow for this scenario is shown in Figure 6.4.

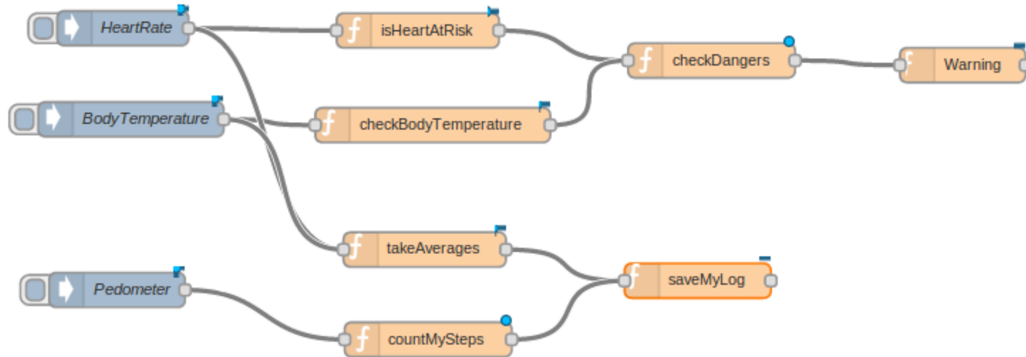


Figure 6-4. Sport Tracker System

In this scenario, the heart rate sensor has the id equal to 1, the body temperature sensor's id is 2 and the pedometer's id is 3. The implemented queries for each node are given in Table 6.3.

Table 6.3. Queries for Scenario 3

Node Name	Query	Node Type
Heart Rate	Topic, 1	Sensor
Body Temperature	Topic, 2	Sensor
Pedometer	Topic, 3	Sensor
Is Heart At Risk	Compare, ID, 1, Operation, >, Threshold, 85	Threshold
Check Body Temperature	Compare, ID, 2, Operation, >, Threshold, 20	Threshold
Check Dangers	Compare, ID, 1, Operation, >, Threshold, 85 Compare, ID, 2, Operation, >, Threshold, 20	Threshold
Take Averages	-	Average
Count My Steps	-	Aggregate
Warning	@sensorquerytest	Twitter
Save My Log	-	HBase

6.3 Scalability Experiments & Results

The scalability of the proposed architecture is investigated with two different environment settings; one slave node and four slave nodes. The average latencies for different number of topologies/queries are measured. Latency is defined as the difference between the timestamp when Kafka spout emits tuple and the timestamp when tuple complete topology acknowledgment. For each test, queries are executed for 10 minutes and the average latencies (in milliseconds) of topologies are calculated. We performed 5 runs for each scenario and average latencies are computed for each scenario.

6.3.1 One Slave Node

In the first set of experiments, the test environment has one master one and one slave node. In other words, we have only one node to process queries. The test results show the average latency for different number of topologies/queries. The number of topologies/queries is varied between 1 and 12.

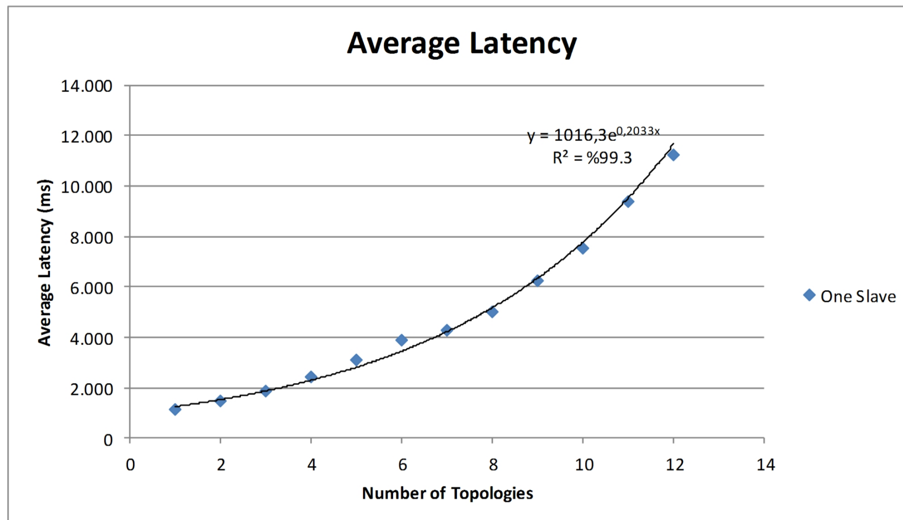


Figure 6-5. Average Latency with One Slave Node for Scenario 1

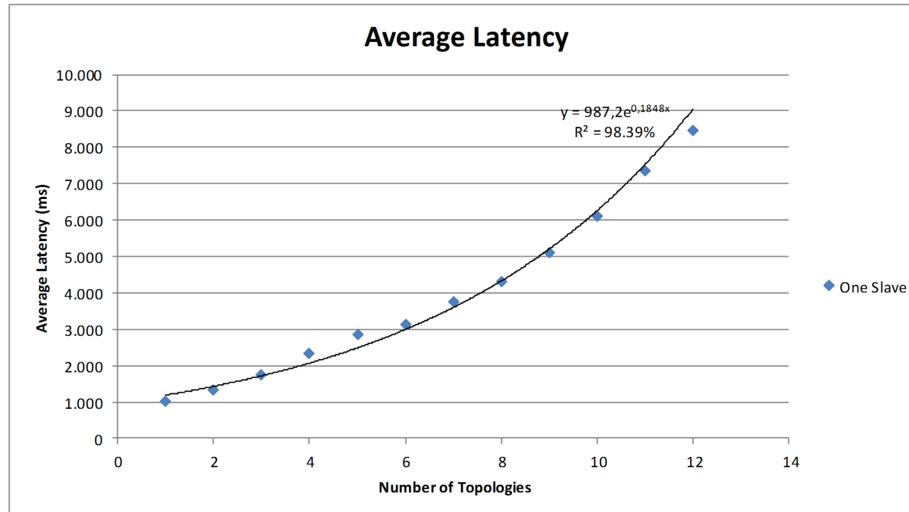


Figure 6-6. Average Latency with One Slave Node for Scenario 2

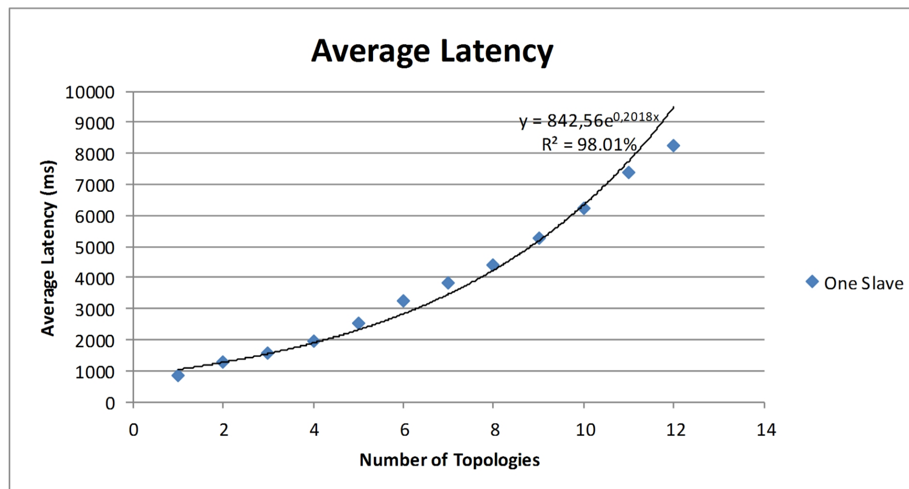


Figure 6-7. Average Latency with One Slave Node for Scenario 3

In this set of experiments, we observed that the average complete latency of the queries is increasing exponentially when the workload is increased. The test result is analyzed by adding exponential trend line and calculating R^2 on the charts in order to observe the exponential increase with the number of queries. The exponential trend line is calculated using the formula;

$$y = c^{ebx} \text{ (Equation 6.2)}$$

where “c” and “b“ are constants and “e” is the base of the natural algorithm.

The R^2 values of the trend lines are calculated in order to analyze how close the data of the charts to the fitted trend line [42]. The calculated R^2 value for scenario 1 with one slave node is 99.3%, 98.39% for scenario 2 and it is 98.01% for scenario 3. These R^2 values show that the exponential trend line fits for these charts. Hence, the proposed system is not able to process more queries with single slave node.

6.3.2 FourSlave Nodes

In the second set of experiments, the environment has one master one and four slave nodes. The test results given in Figure 6-8, Figure 6-9 and Figure 6-10 show the average latency for different number of topologies/queries. The number of topologies/queries is varied between 1 and 40.

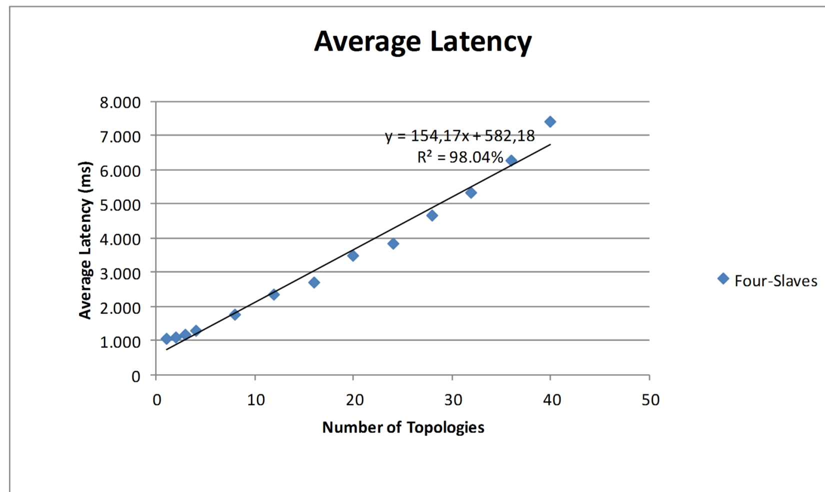


Figure 6-8. Average Latency with Four Slave Nodes for Scenario 1

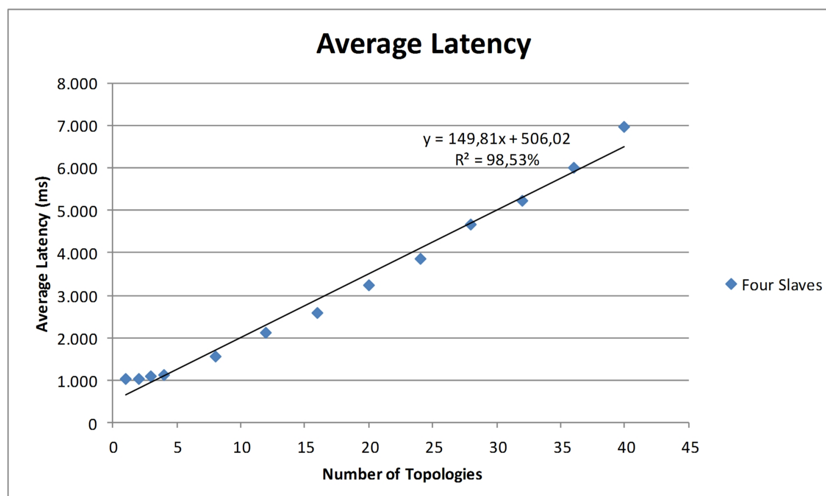


Figure 6-9. Average Latency with Four Slave Nodes for Scenario 2

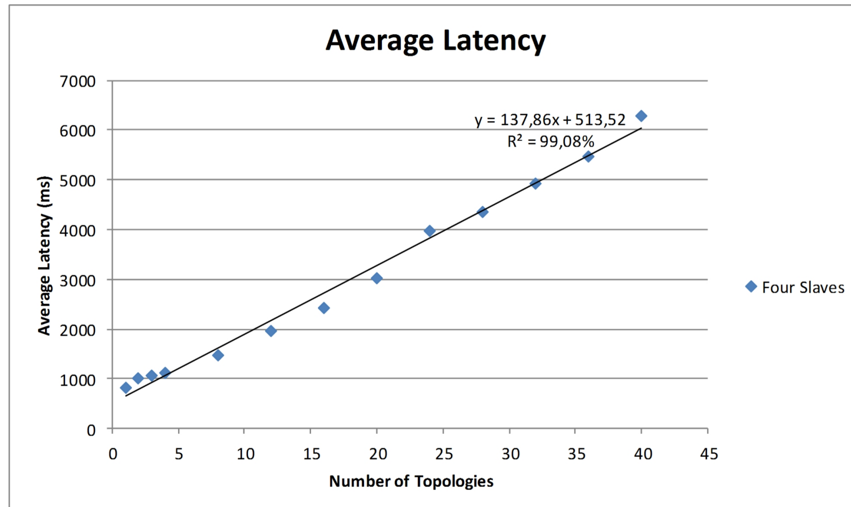


Figure 6-10. Average Latency with Four Slave Nodes for Scenario 3

As it can be seen from Figures 6.8, 6.9 and 6.10, the latency starts to increase after fourth topology because workload of each node is rising slowly as each node starts to execute more than one topology/query after the fourth one.

The main purpose of the second set of experiments is to prove the scalability of the proposed architecture. A linear increase with number of queries is an indication of scalability of the architecture. Thus, the test result is analyzed by adding linear trend line and calculating R^2 on the chart. The linear trend line is calculated using the formula;

$$y = mx + b \text{ (Equation 6.3)}$$

where "m" is the slope and "b" is the intercept.

The calculated R^2 value for scenario 1 with one slave node is 98.04%, it is 98.53% for scenario 2 and 99.08% for scenario 3. This R^2 value shows that the linear trend line fits for this chart. Hence, the proposed system is able to process more queries by increasing the number of slave nodes.

We have also evaluated the scalability of the architecture with scenario specific experiments. We also evaluated the scalability of the proposed architecture with the mixture of the defined scenarios. In this experiment, the number of topologies varies between 3 and 39. We have used four slave nodes and the number of topologies for each scenario is increased one by one. This test results are also analyzed by adding linear trend line on the chart. The calculated R^2 for this experiment is 98.95%. The test results are shown in Figure 6-11. In these experiments, we also observed that there is a relationship between the number of threads and the average complete latency. In scenario-1, 21 threads, 18 threads for scenario-2 and 11 threads for scenario-3 worked in parallel. As it can be seen from the test results given in Figure 6-8-Figure 6-10, the scenario-1 has the highest latency and scenario-3 has the lowest latency for each test. This is because; the workload in the CPU of slave nodes varies for different number of threads.

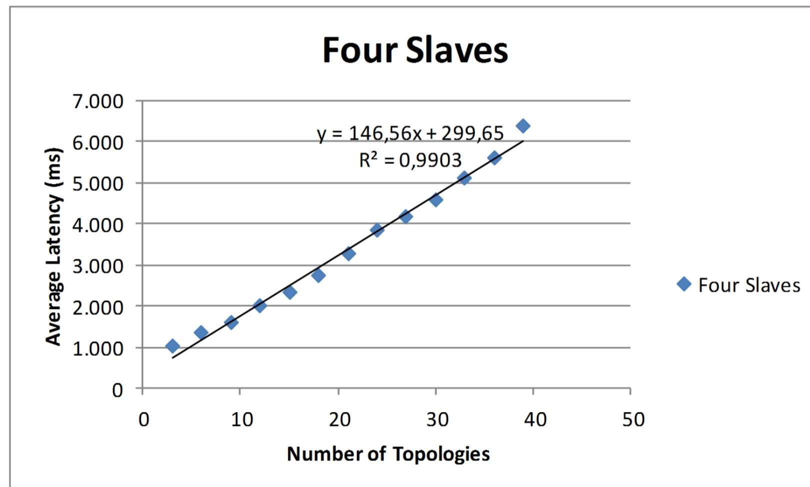


Figure 6-11. Average Latency for the Mixture of Scenarios

The comparison of average latencies for scenario-1 with one slave node, two slave nodes, three slave nodes and four slave nodes is shown in Figure 6-12. In this experiment, the number of topologies varies between 1 and 24 (except for the experiment with one slave node as a single slave node is not able to process more than 16 topologies at the same time). As it can be seen from the figure, the performance of the proposed architecture can be improved by increasing the number of slave nodes. In other words, it is possible to distribute the workload evenly to slave nodes and improve the performance by increasing the number of slave nodes. This is because the load on shared computing resources can be reduced by adding new slave nodes.

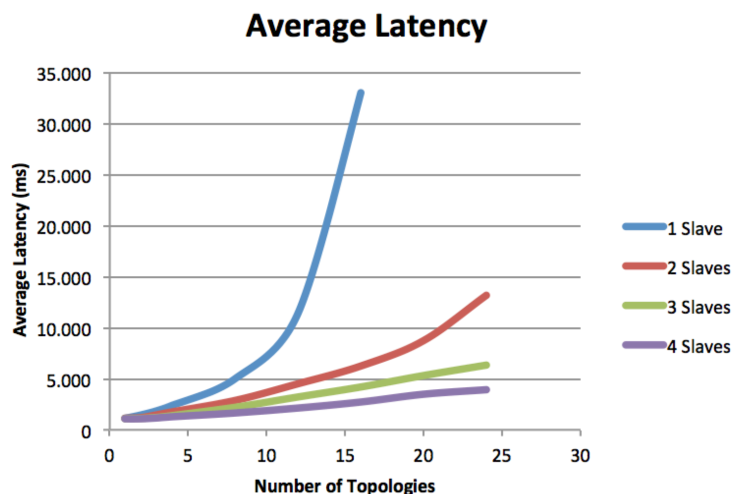


Figure 6-12. Performance Comparison of Slave Nodes

6.4 Comparison of Alternative Data Distribution Methods

The straightforward approach to distribute sensor readings is forwarding all sensor readings directly from master node to all processing units. The straightforward

approach is depicted in Figure 6.13. But this causes unnecessary data transmission between Master node and Slave nodes. Therefore, we have designed an architecture specific message distribution module to improve the performance. In our approach, the sensor readings are sent to slave nodes at first then slave nodes forward sensor readings to processing units. The proposed approach is depicted in Figure 6.14.

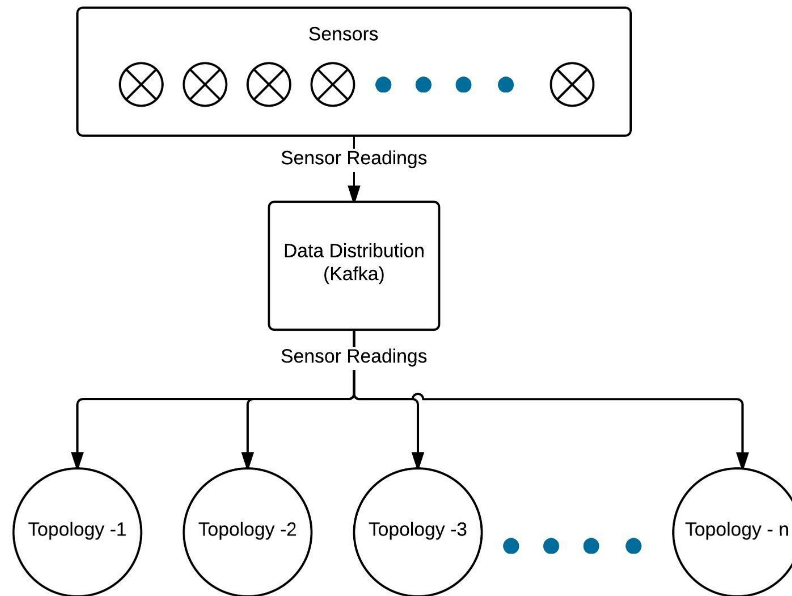


Figure 6-13. Straightforward Approach

In this set of experiments, two basic sensor reading distribution scenarios are defined. In scenario 1, each processing unit in a slave node subscribes to different topics. In scenario 2, each processing unit in a slave node subscribes to the same topics.

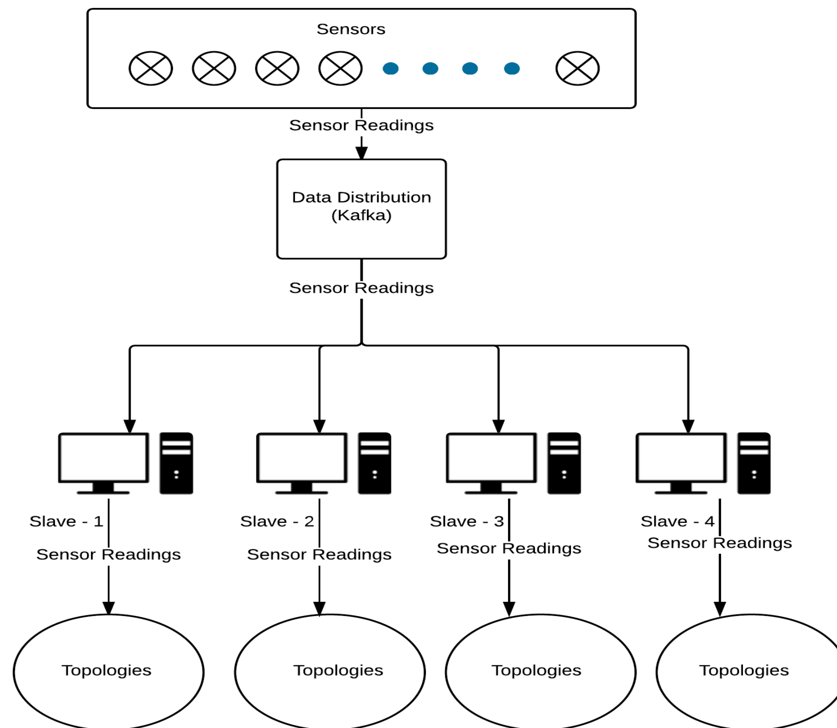


Figure 6-14. Proposed Approach

The results given in Figure 6.15 show that the average latency for different number of topologies for the straightforward approach and the improved approach. The number of topologies is varied between 1 and 40 in order to analyze the factor that affects the performance of a message distribution mechanism. The total number of produced messages for each topic is same. As it can be seen from the test results in Figure 6.15, the latencies of both approaches are close to each other. This is because each system sends the same number of messages to processing units. The latency discrepancy between scenarios is mainly caused by the workload difference on Zookeeper and Kafka servers. The straightforward approach uses only single broker that runs on Master node to distribute all sensor readings to running topologies. In the proposed approach, slave nodes also run local Zookeeper and Kafka servers in addition to the servers that run on Master node to distribute sensor readings to running topologies. That is, in the proposed approach, the workload on the Zookeeper and Kafka servers that run on Master node is less than the straightforward approach.

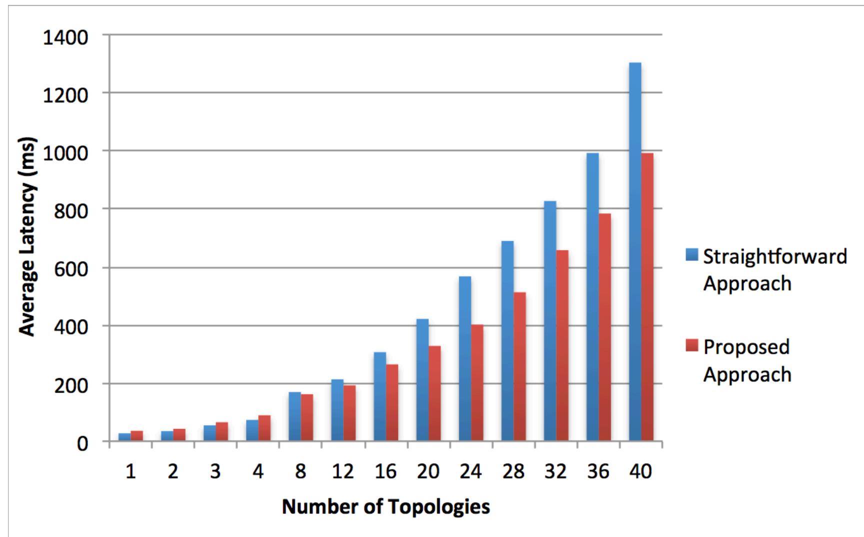


Figure 6-15. Messaging System Test Results-1

In the second set of experiments, each processing unit in a slave node subscribes to the same topic. As it can be seen in Figure 6.16 the latency of the straightforward approach is almost the same with scenario 1. This is because it sends the same number of messages to processing units. However, the average latency of the proposed approach reduces drastically as the number of messages that is affected by network latency is decreased. In other words, the straightforward approach sends a reading of sensor to slave nodes more than one time but the proposed approach distributes each sensor readings to slave nodes exactly once.

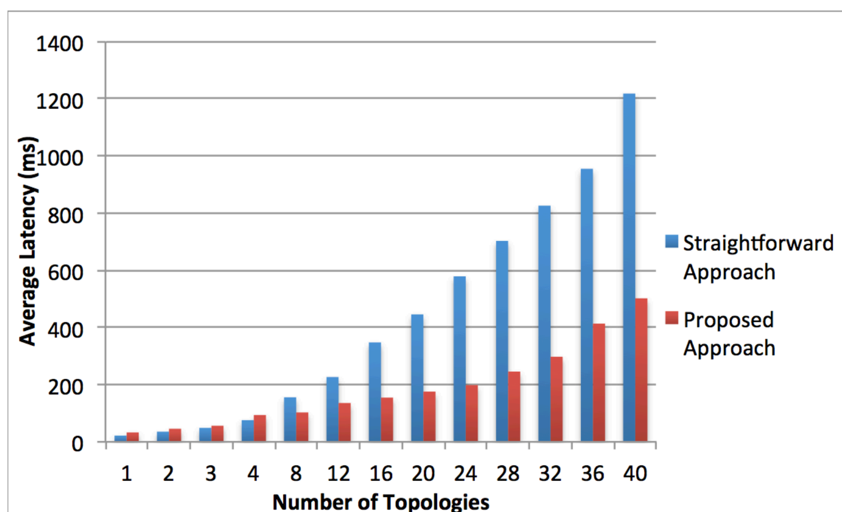


Figure 6-16. Messaging System Test Results-2

CHAPTER 7

CONCLUSION

In this thesis study, a cloud based distributed architecture to process continuous queries is proposed. In this architecture, the open source big data technologies; Storm, Kafka, HBase and Zookeeper are utilized to create a scalable system. The Node Red is also utilized to construct a query via drag and drop visual interface. The utilized technologies are especially designed for distributed computing environment to solve big-data problems. Thus, the proposed architecture inherently supports a distributed computing environment. The utilized big-data tools are also relatively new and developing technologies. In the literature, there are some studies [32, 33] to solve different kind of problems with some combination of these tools. However, there is no such attempt to solve processing continuous queries in real time, to the best of our knowledge.

In the literature, there are some proposals to process continuous queries. However, as mentioned in Section 2, these studies have several shortcomings such as: they do not support distributed infrastructure, they just employ polling based queries and they target use-case specific applications. There is no study that aims running continuous queries in real time, supporting distributed infrastructure and allowing users to define flexible queries. In this thesis study, we proposed an architecture that provides all of these important features.

Thus, the main contributions of this thesis study are proposing an architecture to process complex events in real time and in a distributed computing environment, proposing a graph based definition of complex events and demonstrating the scalability of the proposed architecture by conducting several experiments. We have proposed a directed graph based query definition model to guide users in defining flexible queries. The proposed framework and the query definition model can be applied to various applications in IoT domain.

The prototype implementation is used for the four-leg intersection roads management system, smart home and sport tracker use case scenarios explained in Section 3 to demonstrate the feasibility of the proposed approach.

We have conducted several experiments to assess the scalability of the proposed architecture. According to the results of the experiments, increasing number of queries in real time increases the latency linearly. In other words, it is possible to distribute the workload equally to slave nodes and improve the performance of topologies by increasing the number of slave nodes. We also compared the proposed data distribution module with the straightforward approach. According to the results of the experiments, the straightforward approach sends sensor readings to the slave nodes more than one time, however, the proposed data distribution module forwards each sensor readings to slave nodes exactly once. In other words, the proposed data distribution module reduces the network latency and increases the overall performance of the system.

In big data domain, there are different kind of processing models that are batch, micro-batch and stream processing. In the design phase of the architecture of this thesis study, we have also discussed these three models. The Hadoop MapReduce for batch processing, Spark Streaming for micro-batch processing and Storm for stream processing are analyzed. The Hadoop MapReduce and Spark have a very strict programming model so it is not easy to codify every algorithm as a MapReduce application. It works well for the operation of batch data processing but it is not suitable for handling streaming data. Moreover, it can be time consuming even querying data sets for simple statistical computation. In this thesis study, we aimed to process queries in a real time fashion. Therefore, we preferred utilizing Storm in our architecture, because its performance is better at processing streaming data. Moreover, the graph based topology structure of Storm is similar to our query design approach.

There exist some limitations regarding the prototype implementation. The most significant point that needs an improvement in the query definition model is that the users should have deep knowledge about the query parameters. In other words, each operation has strict syntax to deploy operations properly. Therefore, we are planning to improve our query definition interface to guide users. However, the improvements and advancements of this prototype implementation are left for a future study. Another limitation of this thesis study is the lack of the support for security and confidentiality of the user and the sensor readings. The sensor reading distribution is built upon publish/subscribe protocol with Kafka. Although Kafka supports authorization and encryption across brokers, some additional improvements such as SSL authentication may be added. Another important limitation of this thesis study is about data distribution module. Each slave node runs on local server, they only communicate with master node to subscribe/unsubscribe Kafka topic operations. Therefore, it is hard to handle failures in the servers. In other words, it is not easy to manage and control all of the slave nodes when we increase the number of slave nodes.

We are planning to extend our query-processing module to cover variety of application domain such as analyzing Twitter data. In the prototype implementation,

the distribution of the topologies to slave nodes is under the responsibility of default Storm scheduler. Nevertheless, we are planning to implement our own scheduler for Storm to distribute topologies according to subscribed sensor ids', in order to decrease the latency of the sensor readings to processing units. We are also planning to define a mathematical model to estimate the average latency of the defined queries to allow users to determine the number of slaves required to achieve a given latency level. Moreover, we are planning to enlarge our experimental environment to analyze the proposed architecture more detailed by increasing number of slave nodes. The impacts of Zookeeper, Kafka and network traffic between the slave nodes on the performance of the proposed system architecture can be analyzed.

REFERENCES

[1] Gartner, Technology Research, Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020 [Online] http://www.gartner.com/newsroom/id/2636073 (Last accessed: 30 June 2015)

[2] MapReduce, Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

[3] Apache Storm, https://storm.apache.org/ (Last accessed: 30 June 2015)

[4] Apache Hbase, http://hbase.apache.org/ (Last accessed: 25 June 2015)

[5] Apache Kafka, http://kafka.apache.org/ (Last accessed: 28 June 2015)

[1] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. ACM SIGMOD Workshop on Networking Meets Databases, page 6, 2011.

[2] Node Red, http://nodered.org/ (Last accessed: 27 June 2015)

[3] Zookeeper, http://zookeeper.apache.org/ (Last accessed 27 June 2015)

[4] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In USENIX Annual Technical Conference (Vol. 8, p. 9), June 2010.

[5] Spark Streaming, https://spark.apache.org/streaming/ (Last accessed: 25 June 2015)

[6] Hadoop, https://hadoop.apache.org/ (Last accessed: 29 June 2015)

[7] BBVA Innobation Center, “Big Data:Where we at?,” [Online]. Available:https://www.centrodeinnovacionbbva.com/en/magazines/innovationedge/ publications/20-big-data/posts/147-big-data-where-we-at. (Last Accessed 22 June 2015)

[8] S4, http://incubator.apache.org/s4/ (Last accessed: 25 June 2015)

[9] Storm Trident, https://storm.apache.org/documentation/Trident-tutorial.html (Last accessed: 25 June 2016)

[10] Apache Spark, https://spark.apache.org/ (Last accessed: 25 June 2015)

[11] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on (pp. 1-10). IEEE.

[12] Node.js, https://nodejs.org/ (Last accessed: 25 June 2015)

[13] Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002, June). Models and issues in data stream systems. In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 1-16). ACM.

[14] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[15] L. Liu, C. Pu, and W. Tang. Continual queries for Internet scale event-driven information delivery. *IEEE Trans. On Knowledge and Data Engineering*, 11(4):583–590, Aug. 1999.

[16] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390,May 2000.

[17] Yao, Y., & Gehrke, J. (2003, January). Query Processing in Sensor Networks. In CIDR (pp. 233-244).

[18] Madden, S. R., Franklin, M. J., Hellerstein, J. M., & Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. ACM Transactions on database systems (TODS), 30(1), 122-173.

[19] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., & Zdonik, S. B. (2003, January). Scalable Distributed Stream Processing. In CIDR (Vol. 3, pp. 257-268).

[20] Babu, S., & Widom, J. (2001). Continuous queries over data streams. ACM Sigmod Record, 30(3), 109-120.

[21] A. Arasu, S. Babu, J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Technical Report, Nov. 2002. dbpubs.stanford.edu:8090/pub/2002-57.

[22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In Proc. Conf. on Innovative Data Syst. Res, 2003, pp. 245,256.

[23] Esper. http://esper.codehaus.org/ (Last accessed: 25 June 2015)

[24] Drools. http://www.drools.org/ (Last accessed: 25 June 2015)

[25] Dash, S. K., Sahoo, J. P., Mohapatra, S., & Pati, S. P. (2012). Sensor-cloud: assimilation of wireless sensor network and the cloud. In Advances in Computer Science and Information Technology. Networks and Communications (pp. 455-464). Springer Berlin Heidelberg.

[26] Alamri, A., Ansari, W. S., Hassan, M. M., Hossain, M. S., Alelaiwi, A., & Hossain, M. A. (2013). A survey on sensor-cloud: architecture, applications, and approaches. International Journal of Distributed Sensor Networks, 2013.

[27] Rios, L.G and Diguez, J.A.I "Big Data Infrastructure for Analyzing Data Generated by Wireless Sensor Networks" in IEEE International Congress on Big Data, 2015. p. 816-823.

[28] Lim, H. and Babu, S. “Execution and Optimization of Continuous Windowed Aggregation Queries” Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on. p. 303-309

[29] Armbrust, Michael, et al. "A view of cloud computing." Communications of the ACM 53.4 (2010): 50-58. APA

[30] Groovy, http://groovy.codehaus.org/ (Last accessed: 25 June 2015)

[31] XenServer, http://www.xenserver.org/ (Last accessed: 25 June 2015)

[32] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J. H., ... & Zdonik, S. B. (2005, January). The Design of the Borealis Stream Processing Engine. In CIDR (Vol. 5, pp. 277-289).

[33] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., ... & Culler, D. (2005). Tinyos: An operating system for sensor networks. In Ambient intelligence (pp. 115-148). Springer Berlin Heidelberg.

[34] Bonnet, Philippe, Johannes Gehrke, and Praveen Seshadri. "Querying the physical world." *Personal Communications, IEEE* 7.5 (2000): 10-15.

[35] Twitter, https://twitter.com/ (Last accessed: 25 June 2015)

[36] Maven Assembly Plugin, https://maven.apache.org/plugins/maven-assembly-plugin/assembly.html (Last accessed: 27 July 2015)

[37] Jin, R., Chen, W., & Simpson, T. W. (2001). Comparative studies of metamodelling techniques under multiple modelling criteria. Structural and Multidisciplinary Optimization, 23(1), 1-13.