THESIS

LEVERAGING STREAM PROCESSING ENGINES IN SUPPORT OF PHYSIOLOGICAL DATA PROCESSING

Submitted by

Sitakanta Mishra

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2018

Master's Committee:

Advisor: Shrideep Pallickara

Sangmi Pallickara
Chandra Venkatachalam

ABSTRACT

LEVERAGING STREAM PROCESSING ENGINES IN SUPPORT OF PHYSIOLOGICAL DATA PROCESSING

Over the last decade there has been an exponential growth in unbounded streaming data generated by sensing devices in different settings including the Internet-of-Things. Several frameworks have been developed to facilitate effective monitoring, processing, and analysis of the continuous flow of streams generated in such settings. Real time data collected from patient monitoring systems, wearable devices etc. can take advantage of stream processing engines in distributed computing environments to provide better care and services to both individuals and medical practitioners.

This thesis proposes a methodology for monitoring multiple users using stream data processing pipelines. We have designed data processing pipelines using the two dominant stream processing frameworks – Storm and Spark. We used the University of Queensland's Vital Sign Dataset in our assessments. Our assessments contrast these systems based on processing latencies, throughput, and also the number of concurrent users that can be supported in a given pipeline.

ACKNOWLEDGEMENTS

I am a true believer of God. I achieved this great milestone of my life with the blessings of God. I bow down before the creator and I am very grateful for the grace it bestowed on me.

I always appreciate the stand of my parents in my life. I accomplished my ambition of pursuing my Master's degree because of my father, Dr. Shashikanta Mishra and my mother, Swapna Dwibedy. I am thankful to my sisters Soumya Padhy and Snigdha Mishra for their immense love and support. I would also like to thank my brother-in-laws, Dipty Ranjan Padhy and Mrutyunjaya Panda and my paternal and maternal grandmothers Basanti Mishra and Kamala Kumari Dwivedy for their affection and motivation.

I am heartily grateful to my advisor Dr. Shrideep Pallickara for giving me the opportunity and for believing in me. His insightful guidance and constant encouragement made me complete this thesis successfully. I would like to express my gratitude to my committee members Dr. Sangmi Pallickara and Dr. Chandra Venkatachalam for their valuable inputs and cooperation in completing this work. I would like to acknowledge the Computer Science department and Graduate School of Colorado State University for the great experience and creating an extraordinary learning atmosphere.

I am much obliged to my Guruji Rahul Parmar and Gurumaa Anjali Parmar for their kindness and inspiration. They made me build my self-confidence and self-esteem by teaching me to realize my own self. I would like to thank all of my family members and friends for their tremendous support and motivation towards achieving this goal.

Last but not least, thank you to the creators and maintaners of LaTeX for creating a fantastic typesetting tool.

DEDICATION

I would like to dedicate this thesis to my late paternal grandfather Shri Bishnu Mishra and late maternal grandfather Shri Niranjan Dwivedy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

There has been a massive flow of unstructured streaming data from sensing devices in the Internet of Things (IoT) settings. This phenomenon is gaining momentum in fast pace because of its ability of capture real time data [1]. The number of devices connected to the Internet has increased significantly in several applications such as the smart cities, smart grids and buildings, smart health care systems, global flight monitoring systems etc. The gigantic growth of heterogeneous data from these diverse devices benefits different types of users in decision making by transforming the collected data into valuable information. Stream processing frameworks performs high velocity data processing for the mission critical systems with operational efficiency, reduced cost and diminished risk [2].

Physiological data processing has become one of the important use cases of mission critical data analysis with the rapid development of health care industry. The large amount of health care data includes continuous patient monitoring data, electronic health and clinical reports, diagnostic reports, medical images, pharmacy information etc. [3]. In addition to that, physiological sensors used in the wearable devices monitor patients remotely which creates opportunities for the researchers to extract knowledge by analyzing these data continuously. With the advent of distributed computing in big data analytics, insightful decisions can be made by understanding these data patterns and the associated relationships between them [4] [5].

The Big data computing can be done as batch-oriented and real-time stream computing. Batch computing is efficient in processing huge volumes of data, but it has the potential of introducing significant amount of delays and fails to process unbounded streaming events effectively. However, stream computing emphasizes on the velocity of data processing and it involves continual input and outcome of data [6]. The Stream processing engines receive, store, index and process these massive streams of spatio-temporal real time data which gets generated by millions of users [7]. The streaming frameworks are known for providing high throughput and low latency while performing

data processing in parallel. Hence, the exploration of distributed stream processing engines is of paramount importance in order to provide better health care services to individuals and medical personnels.

1.1 Research Challenges

Building data pipelines to perform predictive analytics and achieving high throughput and low latency in physiological data processing involves several challenges which are described below.

- **Complexity of Data:** Continuous flow of real time events from different physiological sensors constitute complex patterns. It is one of the important technological challenges to process these data in near real time to apply them in decision making and to extract insightful information.

- **Arrival Rate:** The large volume of real time physiological data arrives at high speed from the patient monitoring systems. These streaming events has a fundamental requisite of being processed with minimal latency and high throughput.

- **Scalability and Concurrency:** In the mission-critical systems, the underlying data format might change because of the addition of a new sensor nodes. The stream processing engines should be able to handle these scenarios with minimal execution delays. The parallelization in the distributed stream computing depends with the use case. The data partitioning scheme, size of the thread pool, memory requirements in the executors, number tasks etc. should be configured wisely to achieve good performance.

- **Predictive Analytics:** Predicting physiological information on the streaming data benefits in the early detection of diseases and diagnostics. Training models on the historical streaming data does not guarantee accurate results and introduces complexities. Hence, online prediction on the streaming data is a major research challenge.

1.2 Research Questions

A holistic solution is required to address the challenges described above. The following research questions will be explored as part of this thesis.

1. How can we leverage stream processing engines in support of physiological data stream processing?

2. How can we construct data processing pipelines that are suited for the mission critical nature of such processing?

1.3 Approach Summary

This section provides an overview of the design approach used in this work to overcome different research challenges described above and to reasonably answer the research questions. In the proposed system, Apache open source frameworks are used to design data pipelines. These data pipelines are responsible for processing and performing online prediction on the data.

The University of Queensland vital sign dataset has been used to validate the proposed system. This dataset contains different vital signs of surgical patients who underwent anesthesia at the Royal Adelaide hospital. There is no streaming API connector provided in the dataset [8] rather the vital sign information are represented in a batch dataset. Kafka producers have been implemented to create the streams from the dataset for multiple users and these streaming messages are published into different topics as vital sign events. Storm and Spark stream processing systems consumes these streams in separate pipelines by subscribing to the topics. Data splitting and pattern matching operations are performed inside the pipelines in order to prepare the data for the online predictions. Different vital signs are predicted using different machine learning models in the online prediction component of data pipelines. These models are trained offline on the historical data in order to avoid incorrect model updation and to achieve better accuracy. The predicted vital signs are stored in the Hadoop distributed file system and Linux file system for visualization purposes.

The performance of different machine learning algorithms on several vital signs are compared in terms of variance score and root mean squared error. The two data pipelines using Storm and

Spark are utilized separately for measuring throughput and latency. Spark performed better than Storm in terms of throughput but at the expense of huge latency. The maximum number of users supported by a single pipeline is also evaluated. As evident from empirical evaluations, the overall performance of Storm is better than Spark which in turn validates the suitability of Storm for physiological stream data processing.

1.4 Thesis Contribution

This thesis presents the approach to process physiological data streams using data pipelines and perform predictive analytics on the streaming data. Specific contributions of this work include:

- Exploration of stream processing engines to support physiological data analysis
- Building data pipelines to achieve high throughput with minimal latency
- Performing online prediction of vital signs using any machine learning models
- Support of stream data processing in distributed cluster environment with maximum parallelization configurations
- Determination of maximum number of users supported by a single pipeline which can be increased linearly
- Contrasting performance of dominant stream processing engines in terms of throughput and latency

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides an overview of the proposed system architecture along with the background information on several open source frameworks used to build the system. Chapter 3 described different methodologies and approaches used to build the data pipelines. The evaluations, experiments and results are presented in chapter 4. Chapter 5 gives an illustration of related work being performed in health stream processing and

other mission critical data analysis. Chapter 7 contains the conclusion of the thesis and future work for extending and refining the current methodologies.

Chapter 2

System Design

This chapter describes the background knowledge required for understanding the data pipelines built in this thesis. Several Apache open source frameworks such as Kafka, Storm, Spark and ZooKeeper are used to design the system for health stream processing. We have used The University of Queensland Vital sign dataset to validate the system. This dataset contains different vital signs of 32 patients who underwent anesthesia under surgical procedures. Kafka is used to create continuous streams of events for multiple patients and these vital sign streams are consumed through separate data pipelines by Storm and Spark stream processing engines respectively. The end goal of the system is to predict vital signs in future. The system can support any machine learning algorithms to perform predictive analytics which is described in detail in the methodology section. The predicted vital signs are stored using Hadoop Distributed File System (HDFS) for visualization purpose. The system architecture and roles of the different open source frameworks used to design our system are discussed as follows.

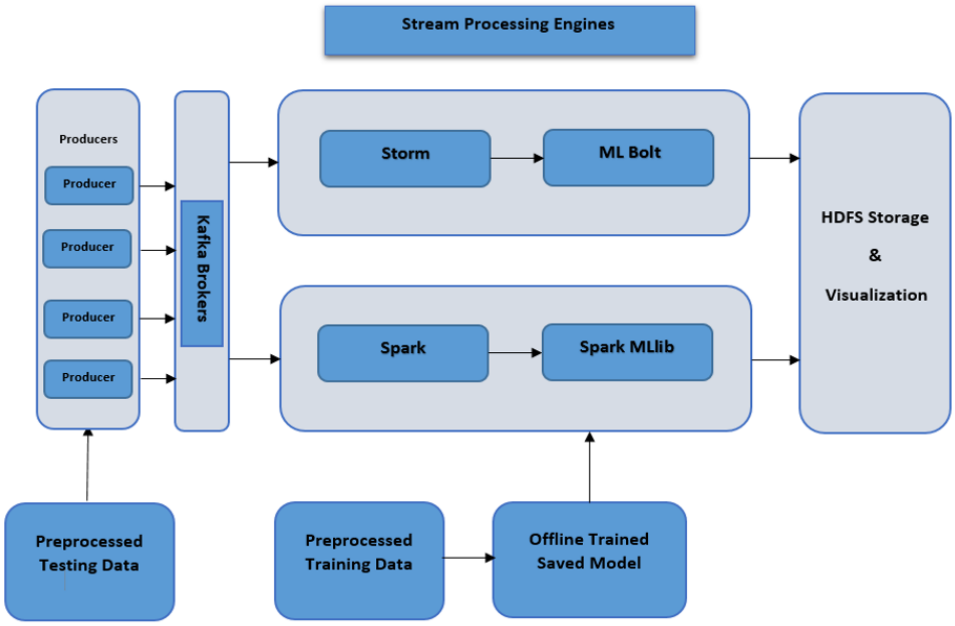

Figure 2.1: Proposed Architecture

2.1 Apache Storm

Apache Storm is an open source distributed stream data processing system which performs complex computation of unbounded streams of data in real time. It is designed for the mission critical systems to provide scalability, fault tolerance, extensibility, easy administration and good performance by keeping all its storage and computational data structures in memory. Storm was initially created by Nathan Marz at BackType and it was improved at Twitter after the acquisition. Storm can support many use cases like real time analytics, online machine learning, continuous computation, distributed remote procedure calls, ETL etc. [9] [6].

2.1.1 Architectural Description

Stream is the core abstraction of Storm's data processing architecture which consists of streams of tuples flowing through topologies which is a directed acyclic graph where the vertices and edges represent the computation components and data flow between these components respectively [9] [10]. Spouts and bolts are the further division of vertices into disjoint sets where spouts are the tuple sources and bolts does the work of processing the incoming tuples and passing them to the next set of bolts downstream respectively for the topology. In the distributed cluster setting, the clients submit topologies to a master node called Nimbus which is responsible for the scheduling, coordination and monitoring of the topologies. There are worker nodes where different parts of the topology gets executed using one or more worker processes which serve as containers inside the host machines. One or more executors are run by each worker process which runs a JVM. Each of the executors run several tasks and the actual work of spouts and bolts are done through these tasks which instead provides parallelism inside the storm. The streaming data is shuffled from the producer to consumer spout/bolt. Different partitioning strategies supported by Storm are described below [9].

1. **Shuffle Grouping:** It partitions tuples randomly.

2. **Fields Grouping:** It hashes on a subset of the tuple attributes.

3. **All Grouping:** In this case, the entire stream is replicated to consumer tasks.

4. **Global Grouping:** In this scenario, the entire stream is sent to a single bolt.

5. **Local Grouping:** In this case, the tuples are sent to the consumer bolts in the same executor.

The state of the cluster is maintained in Zookeeper [11]. A Supervisor is run by each worker node in order to establish the communication with the master node, Nimbus. Storm spawns the instances of spouts and bolts specified by the user in the topology and creates the interconnection for the data flow [9]

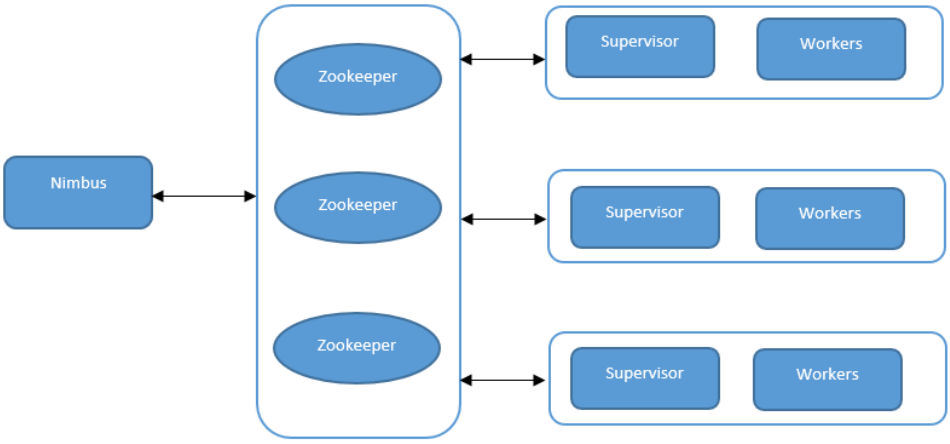

Figure 2.2: Storm System Architecture

Nimbus and ZooKeeper: *Nimbus* acts as a job tracker which is an Apache Thrift service and Storm topology definitions are thrift objects. After a topology has been submitted to *Nimbus*, the state of the topology is stored using a combination of local disks and Zookeeper. Periodic heartbeat messages are used by *Supervisors* to contact to *Nimbus* where they advertise the different topologies they are currently running and the vacancies to run more topologies. *ZooKeeper* manages all the states and coordination between *Nimbus* and *Supervisors* providing resilience to Storm. The workers continue to make progress if the *Nimbus* service fails and also the *Supervisors* restart the workers in case of their failures. New topologies cannot be submitted in case of the failure of

Nimbus and in case of machine failures, the running topologies cannot be reassigned to different machines until the *Nimbus* is revived [9].

Supervisor: *Supervisors* receive assignments from *Nimbus*, initiates workers based on these assignments and also monitors the health of the workers. Three different threads are spawned by *Supervisor* where the main thread is responsible for reading the Storm configuration, initializing *Supervisor*'s global map, creating a persistent local state in the file system and scheduling recurring time events. The event manager thread is responsible for managing the changes in the existing assignment and the process event manager thread is responsible for managing worker processes that run a fragment of the topology on the same node as the *Supervisor*. Different events used in this context are explained below [9] [10].

1. **Heartbeat Event:** It runs in the context of main thread in every 15 minutes and reports the liveness of Supervisor to Nimbus.

2. **Synchronize Supervisor Event:** It runs in the context of event manager thread every 10 seconds.

3. **Synchronize Process Event:** It runs in the context of process event manager thread in every 3 seconds. It reads the worker heartbeats from the local state and classifies those workers as either valid, timed out, not started or disallowed.

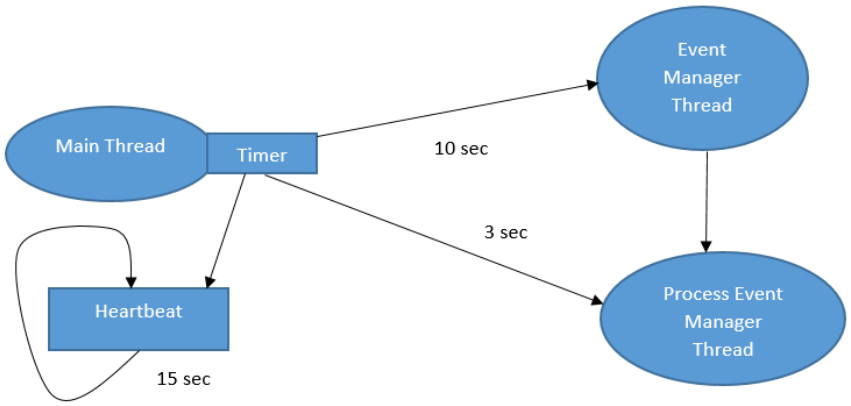

Figure 2.3: Supervisor Architecture in Storm

Workers and Executors: There are two threads to route incoming and outgoing tuples in Storm. The worker receive thread serves as demultiplexing point for all the incoming tuples by listening on TCP/IP port. The tuple destination task identifier is examined by the worker receive thread and the incoming tuples are queued based on it executor. The user logic thread of the executor runs the actual task of spout or bolt instance by taking the incoming tuples from the queue and generates the output by placing the tuples in the out queue. These outgoing tuples are consumed by the executor send thread of the executor and are placed in the global transfer queue. The worker send thread of the Storm sends the tuples to the next worker downstream after examining each tuples in the global transfer queue [9].

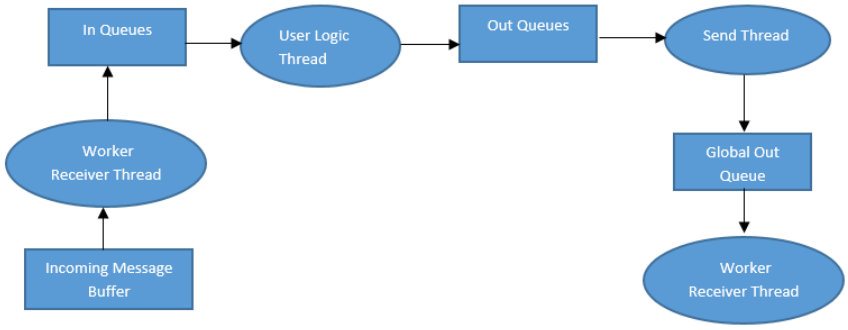

Figure 2.4: Message Flow Inside a Worker Node in Storm

Processing Semantics: Storm provides "at least once" and "at most once" semantics. In case of "at least once" semantics, it is guaranteed that each tuple that is input to the topology will be processed at least once with the support of an "acker" bolt which keeps track of DAG of tuples emitted by a spout. The "at most once" semantics is achieved by disabling the acking mechanism for the topology where each tuple is either processed once or dropped when failure occurs.

2.2 Apache Spark

Most of the systems like MapReduce which are successful in the implementation of large-scale data intensive applications are built around an acyclic data flow model which are not suitable for

iterative machine learning algorithms, interactive data analysis applications etc. Spark supports these applications with working sets while providing better scalability and fault tolerance. Spark introduces an abstraction called resilient distributed datasets (RDDs) which are read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark outperforms Hadoop in many use cases and it can query giga bytes of data with sub-second response time [12]. RDDs are cached in memory across machines and can be reused in parallel operations. Fault tolerance in RDDs are achieved through lineage graphs where the RDDs are rebuilt from the information of its derivation from other RDDs in case of the loss of a partition of the RDD. The implementation of Spark is done in Scala which is a statically typed functional programming language for the Java virtual machine. Scala is chosen because of its combination of conciseness for interactive use and efficiency [12] [13].

RDDs can be created through deterministic operations on either data in stable storage or by performing transformation operations like map, filter and join on other RDDs. Programmers can use these RDDs by performing operations that can return a value to the application or export data to a storage system. These actions are computed lazily on the RDDs the first time they are used in action, so that it can pipeline transformations. RDDs can be persisted in memory or spilled to disk for future use in case of lack of enough RAM space. Users can also specify to store the RDDs only on disk or replicating across machines through persist flag and also persistent priority can also be set on each RDD in order to specify which in-memory data should be spilled to disk first. The immutable nature of the RDDs let the system to run back copies of stragglers or slow tasks. Spark has classified the dependencies between RDDs into two types. In case of narrow dependencies each partition of the parent RDD is used by at most one partition of the child RDD, but in case of wide dependencies multiple child partitions may depend on it [13].

Spark has a driver program which defines one or more RDDs and invokes actions on them. The driver program also connects to a cluster of worker nodes and keeps track of the lineage graph of RDDs. The worker nodes can store RDD partitions in RAM across operations. These workers are long lived processes. In the run time, the driver program of the user launches multiple workers

which read data from the distributed file system and they also can persist these computed RDDs partitions in memory.

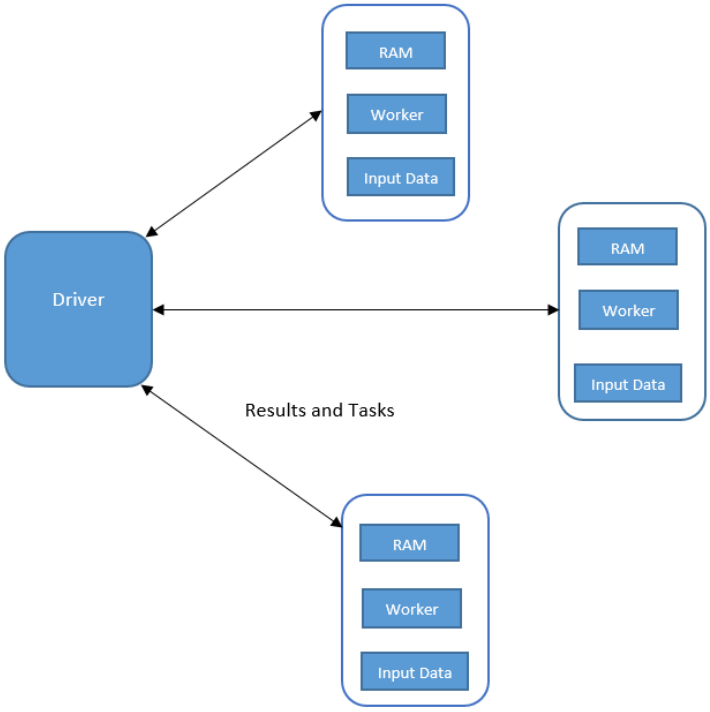

Figure 2.5: Spark Run time

2.2.1 Spark Streaming

Spark streaming lets users intermix streaming, batch and interactive queries seamlessly. The main programming model in Spark streaming is called discretized streams (D-Streams) which provides strong consistency and efficient fault recovery. D-streams are a series of deterministic batch computations on small time intervals which overcome the fault tolerance and consistency in other stream processing systems. Spark streaming allows combining streaming data with historical data in order to make decisions [14]. RDD in-memory storage abstractions are used to rebuild lost data without replication by keeping track of operations needed to be recomputed which eventually reduces the latency. Parallel recovery techniques are used in Spark streaming to recover quickly

from node failure where each node in the cluster works to recompute part of the lost node's RDD [14].

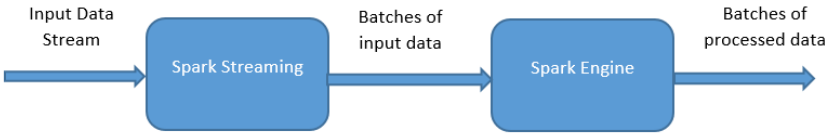

Figure 2.6: Spark Streaming

Discretized Streams (DStreams): The input data received during each batch interval are stored as input datasets. Every input DStreams are associated with a Receiver object which receives data from one or more sources and stores it in the memory of Spark for further processing. This dataset is processed through deterministic parallel operations like map, reduce, groupBy etc. The outputs of the operations and intermediate state is stored in RDDs. These output operations push the data in the DStreams to external systems like the databases or file systems. In order to manipulate the D-Streams, Spark provides stateless operators like map which act independently on each time interval and stateful operators like aggregation over a sliding window, time-skewed joins etc. which operate on multiple intervals and in this scenarios, there is a possibility of intermediate RDDs being produced. To address the problem for fault recovery of nodes, Spark streaming uses lineage graphs to recompute the lost partitions of the RDD in parallel. In addition to that the checkpointing of the state of the RDDs are done such as by replicating every fifth RDD to prevent infinite recomputation of the RDDs [14] [15].

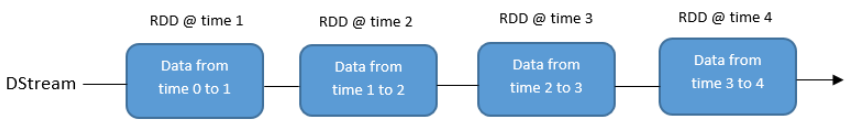

Figure 2.7: DStreams containing data at different time intervals in Spark

MLlib Operations: Machine learning algorithms provided by Spark MLlib can be used in the context of Spark streaming. Spark also provides some streaming machine learning algorithms like Streaming Linear Regression, Streaming KMeans etc. which can learn and apply the model on the streaming data simultaneously. However, majority of the machine learning algorithms are learnt offline using historical data and this model is applied online on the streaming data [15]. This idea is being used in order to address the health streaming use case for this project which will be described in the methodology section.

2.3 Apache Kafka

Apache Kafka is a fast, scalable, fault tolerant and distributed public subscribe messaging system which was initially developed at LinkedIn and then became an open source Apache project in 2011. Kafka provides high throughput, built-in partitioning and replication for which it stands out as a large-scale message processing system as compared to traditional message brokers [16] [17].

In distributed messaging systems, messages are queued asynchronously between client applications and messaging systems. In case of public-subscribe messaging system, messages are persisted in a topic which produced by publishers and consumers can consume to one or more topics by subscribing to them. Kafka is such a system which is suitable for both offline and online message consumption. Kafka is built on top of Zookeeper synchronization service and integrates well with stream processing engines like Storm and Spark. Kafka can handle large number of diverse consumers with low latency message delivery and can perform up to 2 million writes/sec. All the writes in Kafka go to the page cache of the OS (RAM) by persisting all the data to disk which makes it very efficient to transfer data to a network socket [16].

2.3.1 Components and Description:

Kafka cluster constitutes of several components. Different terminologies of Kafka architecture are described below.

1. **Topics:** Topics are streams of messages belong to a particular category which are split into partitions. These partitions are a set of equal sized segment files which can handle arbitrary amount of data. Replicas of these partitions are kept in order to prevent the loss of data [16].

2. **Brokers:** The published data are maintained through broker system in Kafka which can have zero or more partitions per topic. More than one broker constitute Kafka cluster which are used to manage persistence and replication of message data [16].

3. **Publishers:** Producers publish messages to Kafka Brokers which appends the message to the last segment file or partition. These producers can also choose partitions to send messages [16].

4. **Consumers:** Consumers subscribe to one or more topics and read published messages by pulling data out of the brokers [16].

5. **Leader and Follower:** The node responsible for all the reads and writes for the given partition is known as the leader server and the follower node follows instructions from the leader. The follower acts as normal consumer and one of the follower becomes the new leader in case of leader failure [16]

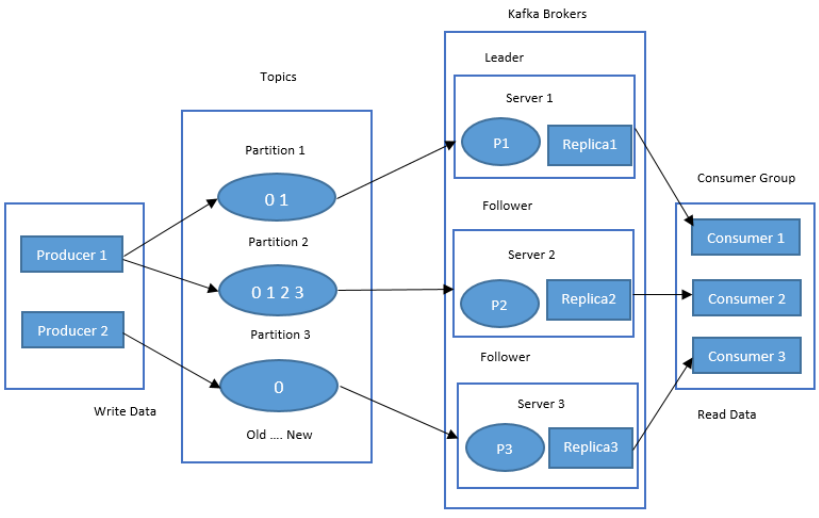

Figure 2.8: Kafka Architecture Diagram

2.4 Apache ZooKeeper

ZooKeeper coordinates between distributed processes through znode registers which are a shared hierarchical name space of data registers like file systems. ZooKeeper provides high throughput, low latency, high availability and strictly ordered access to znodes to its clients. It allows sophisticated synchronization primitives for the clients through strict ordering and it provides better performance through large distributed system [18].

The name space system of ZooKeeper is more like a standard file system, but it is different in the sense that every znode can have data associated with it and the amount of data is limited to the znode. Metadata information such as status information, configuration, location information etc. are stored by the ZooKeeper and it is designed to store smaller pieces of data [18].

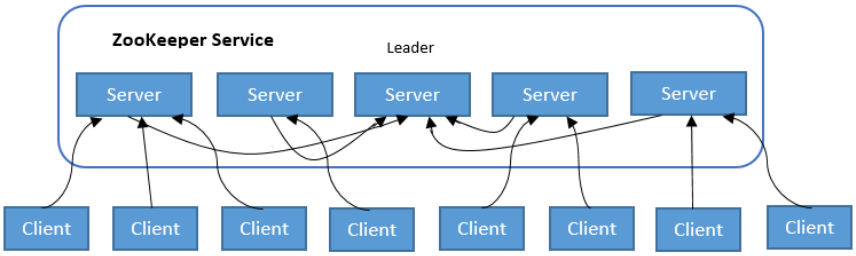

Figure 2.9: ZooKeeper Service

The ZooKeeper service is replicated over a set of machines that comprise the service and these machines maintain in-memory image of the data tree along with the transactional logs and snapshots. All the servers and the clients must know about all the servers in the ZooKeeper service. The clients exchange request-response messages, heartbeat messages and receive watch events by maintaining TCP connection with a single ZooKeeper server. The re-establishment of connection to a new server happens in case of the TCP connection breaks and if the client wishes to connect to another server. Read requests sent by ZooKeeper clients are processed locally at the ZooKeeper server to which the client is connected and the write requests get forwarded to other ZooKeeper servers. Sync requests are similar to write requests, however, write requests go through consensus

before a response is generated. Hence, the throughput of read requests in a ZooKeeper servers increases with the increase in number of servers, but the write request throughput decreases with the increase in number of servers. All the updates in ZooKeeper are totally ordered and in order to reflect the order, ZooKeeper attaches a unique number named as ZooKeeper transaction id (zxid) for each update [18].

Chapter 3

Methodology

This chapter illustrates the implementation, methods, and analytical approaches taken in this research to accomplish the goal of building health streaming data pipelines. This description also includes the different techniques used to preprocess the vital sign data, creation of input streaming events, processing of the streams with the target of achieving high throughput and low latency, offline machine learning model training and online model prediction on the streaming data.

3.1 Dataset

The University of Queensland vital sign dataset [8] is used in this thesis to evaluate the performance of our data pipelines. This dataset contains a collected set of high-quality, high-resolution, multiple-parameter monitoring data suitable for health stream processing research. It includes a wide range of vital signs variables that are commonly monitored during surgery. This dataset provides clinical anesthesia monitoring data from entire surgical case where patients underwent anesthesia [8].

The vital signs data were recorded at the Royal Adelaide Hospital from 32 patients (25 general anesthetics, 3 spinal anesthetics, 4 sedations). In order to capture, time synchronize, and interpolate vital signs data from Phillips IntelliVue MP70 and MP30 patient monitors and Datex-Ohmeda Aestiva/5 anesthesia machines into 10 millisecond resolution samples, software were developed. These data were collected ranging in duration from 13 minutes to 5 hours with a median of 105 minute. In most of the cases, electrocardiograph, pulse oximeter, capnograph, noninvasive arterial blood pressure monitor, airway flow, and pressure monitor data were included, but in few of the cases, data from the Y-piece spirometer, electroencephalogram monitor, and arterial blood pressure monitor are included [8].These vital sign data are saved in four user-friendly file formats such as comma-separated values text files with full numerical and waveform data, numerical parameters recorded in comma-separated values files at 1-second intervals, graphical plots of all waveform

data in a range of resolutions as Portable Network Graphics image files, and graphical overview plots of numerical data for entire cases as Portable Network Graphics and Scalable Vector Graphics files. The complete dataset is available in a public repository and also it has been listed in the Australian National Data Service Collections Registry [8].

3.1.1 Data Preprocessing:

The numerical values recorded in the comma-separated value files with one second intervals are considered to evaluate our use-case. There are 65 features included in the entire dataset. The major challenge faced in this data preprocessing work is the problem of missing parameters. We did not perform any imputation techniques to fill in the missing values in the vital sign measurements, rather the records were dropped from the entire dataset if any feature were detected with missing parameters. We considered dropping the missing values would be statistically significant rather using any imputation techniques such as mean of the entire feature because we suspected that it might lead to superficial results in the predictive analytics.

After dropping the records with missing values, 27 sparse features were selected by inspecting the dataset and avoiding the unnecessary items. These features include the relative time, heart rate, pulse rate, SpO2, etCO2, imCO2, awRR, NBP (Sys), NBP (Dia), NBP (Mean), etSEV, inSEV, etN2O, inN2O, etO2, inO2, ECG, Pleth, CO2, AWP, AWF etc. Pearson correlation co-efficient is calculated between all the featured in order validate the statistical correlation amongst features. A more detailed description of the work is provided in the evaluation section. 28 patient monitoring information are combined to be used as training data for the machine learning models and 2 patient monitoring information are used for the testing of the models. These preprocessed cleaned data are stored separately in comma-separated value files.

3.2 Kafka Producers

The University of Queensland vital sign dataset used for this thesis contains batch data stored in comma-separated value files. It does not provide any streaming API to connect through the

stream processing engines. Hence, we have created stream events by reading through each records of the dataset using Kafka producers which are then consumed by Storm and Spark for further processing.

The Kafka producer code is written in Java programming language. The IP address and port of bootstrap servers or brokers of Kafka cluster have been set in the property settings with a configuration that minimum number of replicas must acknowledge a write for it to be considered as successful. The Kafka producer has been configured with retries set to 0 so that the clients will not resend any record whose sending operation fails with a potentially transient error. Several other parameters such as "linger.ms" is set to 1 millisecond so that any records that arrive between this time duration will be grouped together into a single batched request by the producer. This setting is dependent on the "batch.size" parameter where the producer will linger to accumulate for the partition if the records for a partition is less than 16384 bytes. The producer can buffer records which can wait to be sent to the server, up to a total memory of 33554432 bytes in our configuration setting.

The Kafka producer streams the test data to the stream engines for online prediction of vital signs which is set aside in the data preprocessing step. In order to read the comma-separated value files input streams are created and then the header line is extracted using the Buffered Reader. The CSV file is parsed using this column header and we iterated over each of the records to send it to the Kafka broker. We created a map of column name and value for a record. Finally, a key-value tuple message is created which is sent to the named topic received from the command line on the Kafka broker(s).

3.3 Kafka-Storm Pipeline

Storm subscribes to the Kafka producers or topics using kafka-spout connector to consume the real-time events. In order to throttle the kafka-spout, we have configured the maximum spout pending requirement in the storm framework to 50. ZooKeeper host and port information is provided as an argument during the object creation of broker host for kafka-spout connector along with the

name of the Kafka topics from which storm will consume data. The first ingestion point in our topology is the kafka-spout which is followed by several bots such as "SplitBolt", "PredictBolt" and "ResultBolt". All of these spouts and bolts have been configured with different parallelization parameters with the number of executors and multiple tasks for each of the executors. Shuffle grouping partitioning strategies are used for all the bolts which partitions the tuples randomly. We have executed the storm topology in both standalone and remote mode with multiple number of worker nodes to introduce concurrency.

The tuples consumed by Storm from Kafka topics were not ordered as per our column header and the value of each of the variables are attached with their column number. Hence, they were needed to be processed in an ordered matrix format in order to be used in the online prediction system. In the "SplitBolt", all of these individual records were split, matched with their column name and arranged orderly in a string which is emitted to the "PredictBolt". In the "PredictBolt", all of these records were arranged in the matrix as per the predictor and response variables. The models saved in the pickle files during the offline training process are loaded once in the beginning to predict the vital signs online. These emitted predicted values are stored in the Linux filesystem for visualization in "ResultBolt". Storm framework supports multiple languages. So, the machine learning prediction system is built using Python programming language with the support of scikit learn package and Java programming language is used to build other parts of the topology which includes kafka-spout, split bolt and result bolt. Our system can support any number of machine learning algorithms which can run in parallel and can predict any vital signs at the same time. These results can also be stored into different file systems. A high-level design of our topology is shown in the figure 3.1 below.

3.4 Kafka-Spark Pipeline

In kafka-spark pipeline, a spark context was created with the configuration of non-blocking I/O block transfer service. The spark context is used to load the saved machine learning models in the beginning in order to perform online prediction of vital signs and it is passed as an argument while

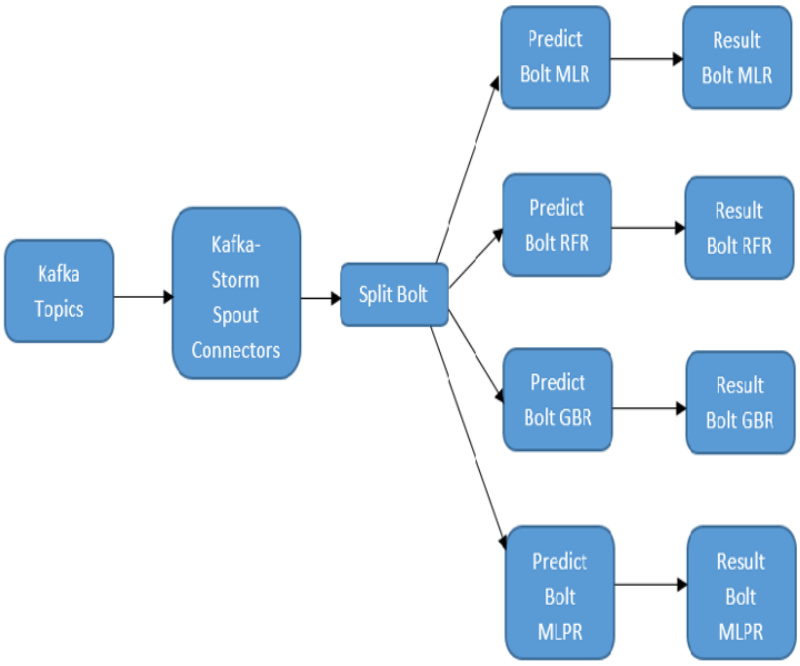

Figure 3.1: Kafka Storm Pipeline

creating the spark streaming context to receive stream events. Spark streaming context is created with a batch interval of 2 seconds and with the check pointing strategy to enable fault tolerance in system. Spark streaming will collect all the events within the 2 seconds time slice of batch interval to form a micro batch and each of these batches gets converted to RDDs. These continuous stream of RDDs are represented as DStreams.

We have followed the Receiver-based approach in Kafka connectors rather the Direct Streaming approach for Spark streaming. The streaming data from Kafka are received through all the receivers in Spark and these are stored in Spark executors to launch jobs to process the data. We have done several configurations to achieve the best possible parallelism in our system. As each of the users publish their data into different topics in Kafka, we have created a map of topics to its associated number of threads. The topic names, number of threads for each of the topics, a group name to receive streaming data for a single Spark job and the Zookeeper host and port information are obtained from the command line. We have created 10 number of DStreams or receivers to consume data in parallel from multiple users of Kafka topics. After creating the Kafka input stream,

we have used the union transformation operation to unify all the streams received from different DStreams. In order to introduce parallelism in our execution, we have repartitioned our unified stream RDD. Then we processed the data to create the required matrix in the right order with the response and predictor variables for online prediction. Finally, the RDDs containing the predicted vital signs are collected from all the executors and stored in the Hadoop distributed file system for visualization. The kafka-spark data pipeline system can support any number of machine learning algorithms which can run in parallel and can predict any vital signs at the same time. The DAG (directed acyclic graph) visualization of the system is shown in figure below.

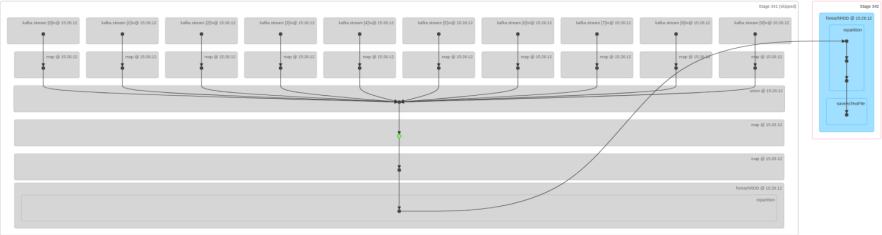

Figure 3.2: Spark DAG

3.5 Machine Learning Algorithms

The system designed in this thesis can accommodate any machine learning algorithms to perform predictive analytics. The University of Queensland vital sign dataset contains a set of continuous variables. Hence, different regression algorithms are considered in order to predict the vital signs. As the data is collected in time series fashion with an interval of 1 second, we have prepared the data in such way that a row of independent variables (x) is mapped with a dependent variable (y) 10 seconds ahead of time. So, the predicted variables will be the vital signs 10 seconds in future. As described above, the records with missing values are dropped during data preprocessing. So, the system tolerates some incorrect predictions which propagates through error. The accuracies and root mean squared errors of different algorithms are compared in the scope of the project for three vital signs which are the heart rate, pulse rate and CO_2 concentration. In an ideal scenario

with no missing parameters, the system can predict any number of vital signs in parallel with less errors. However, this will depend upon the performance of the model chosen during the training time.

The accuracy of the models are calculated by the proportion of the variance in the dependent variable which is predictable from the independent variables. The equation for the coefficient of determination ("R squared") is given as below.

$$R^2 \equiv 1 - \frac{SS_{res}}{SS_{tot}} \tag{3.1}$$

Here, SS_{res} is the sum of squares of residuals and SS_{tot} is the total sum of squares. The root mean squared error is calculated by taking the square root of mean squared differences between the predicted values and the actual dependent variables in the test dataset.

3.5.1 Multiple Linear Regression:

In this regression algorithm, the relationship between two or more independent variables (x) which are otherwise known as explanatory variables and the dependent variable (y) which is otherwise known as response variable is modeled by fitting a linear equation to observed data. The association between the independent and dependent variable is represented by the population regression line which describes the change in the mean response with the change in independent variables. It is assumed that the observed values have the same standard deviation. Let us consider that there are n observations, hence, the model for the multiple linear regression can be formulated as follows.

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ...\beta_p x_{ip} + \epsilon_i \quad for \quad i = 1, 2, ...n \tag{3.2}$$

We normalized the data before training with the multiple linear regression model in order to obtain better accuracy during prediction. This model is used to predict all the three vital signs, i.e. heart rate, pulse rate and CO_2.

3.5.2 Random Forest Regression:

Various sub-samples of the dataset are used to fit a number of classifying decision trees in the random forest meta-estimator. In order to improve the predictive accuracy and control the over-fitting of the estimator, random forest uses averaging techniques. In this method, each tree in the ensemble is built from a sample from the training set and drawn with replacement which is otherwise known as the bootstrap sample. The best split is chosen amongst a random subset of features in this algorithm. The bias of the forest increases slightly because of the random sub-sampling, but the overall model performance is compensated with the decrease in variance because of the averaging techniques. This model is also used to predict all the three vital signs, i.e. heart rate, pulse rate and CO_2. In order to obtain the best performance for our use case, we have set the parameters of number of trees in forest to be 350 and with a maximum depth of 5.

3.5.3 Gradient Boosting Regression:

This technique produces a prediction model in the form of an ensemble of weak prediction models. The decision trees are used as the weak learners and a differentiable loss function is used in this algorithm. A gradient descent procedure is used to minimize the loss of weak learner decision trees when adding the trees which instead reduces the residual loss. In order to improve the final output of the model, the output of the new tree is added to the output of existing sequence of trees. The training of the model is stopped when the loss reaches an acceptable level or it does not improve on a validation dataset. This model is also used to predict all the three vital signs, i.e. heart rate, pulse rate and CO_2. In order to obtain the best performance for our use case, we have set the parameters of number of trees in forest to be 500 with a maximum depth of 4, the minimum number of samples required to split an internal node as 2, a learning rate of 0.01 to shrink the contribution of each tree and the least squared regression loss function required to be optimized.

3.5.4 Multilayer Perceptron Regression:

It is a class of feedforward artificial neural network which consists of at least three layers of nodes. It can distinguish data which are not linearly separable. Each node is known as a neuron which uses an activation function which defines the output of the node on the basis of given input or set of inputs. The input nodes in the MLP do not use any activation function. MLP uses backpropagation supervised learning technique to calculate the gradients needed for the weights to be used in the network. This model is also used to predict all the three vital signs, i.e. heart rate, pulse rate and CO_2.

In order to obtain the best performance for our use case, we have set the parameters of number of hidden layer sizes to 10, activation function for the hidden layers as rectified linear unit, the weight optimizer as stochastic gradient, regularization term parameter as 0.001, mini-batch size for the stochastic optimizer is set to the minimum between 200 and the sample size, learning rate as constant, initial learning rate as 0.01which controls the step size in updating the weights, the exponent of inverse scaling learning rate as 0.5. Some other parameters such as maximum number of iteration is set to 1000, samples in each iteration are allowed to shuffle, the seed used by the random number generator is set to 9, tolerance for the optimizer as 0.0001, momentum for the gradient descent update as 0.9 and the proportion of the training data to set aside as validation set for early stopping is set to 0.1.

3.5.5 Offline Model Training:

We have chosen to perform offline training of the models in this thesis because of several reasons. First, in stream processing systems, the events are either processed in micro batches or a single record at time. We can collect more samples from the stream of events by increasing the batch intervals, but the processing latency increases with the increase in batch intervals and our aim of the project was to reduce the processing latency. Second, the machine learning models perform better if they are trained with large sample size of historical data, but training in micro batches may

raise questions in the accuracy of the models. Third, we suspected that updating hyper parameters in order to minimize the error propagation in the small batches of data could be tricky.

We have used the scikit learn packages for the Kafka-Storm data pipeline in Python programming language to train all the machine learning models described above. The Spark MLlib packages are used in Scala programming language to train all the machine learning models for the Kafka-Spark data pipeline. The trained models are saved as pickle files for the Kafka-Storm data pipeline and in Kafka-Spark data pipeline the metadata information of the trained models are saved in parquet file for the future use. The predictions are performed online in both of the data pipelines using these saved models.

Chapter 4

Evaluation

In this chapter, we demonstrate a series of evaluations carried on Storm and Spark stream processing engines to contrast their performance in terms of throughput and latency. We analyzed a set of individual performance benchmarks for Spark in order to observe the relation between different parallelization configurations and micro batch sizes. We also studied the pattern of throughput and latency in Storm with different concurrent settings and the number of events produced per unit time in Kafka is measured with an increase in the number of users. The performance of several machine learning algorithms used for the purpose of online predictions are described in this section. The hardware and software experimental setup used for the evaluation is illustrated prior to the discussion on experiments.

4.1 Experimental Setup

The Apache Storm and Spark cluster used in this experiment consists of 18 physical machines connected through a local area network with 12.48 Mbps download speed and 31.80 Mbps uploading capability. Each node in the cluster is an Intel(R) Xeon(R) 2.60GHz with 16 CPUs, 8 cores per socket and 30 GB of memory.

We have used the storm version 1.0.4 with the support of exactly once semantics which disables the acknowledgments to reduce the latency. A single physical machine is used as the nimbus seed and 18 worker nodes as the supervisors in the cluster. Spark version of 1.6.1 is used which runs in association with Hadoop version of 2.7.3. In spark, 18 slave nodes or executors are used with 3 GB of executor memory and 2 GB of driver memory. All the experiments in Storm and Spark are conducted in cluster mode. We have used Kafka version of 2.11-0.11.0.0 with a 3 broker cluster running on 3 different physical machines. All the topics are created with 3 partitions and the level of replication as 2 for running all the experiments. The states of Kafka and Storm cluster are maintained in the ZooKeeper server with a version number of 3.4.10. We have used a single

physical machine to run the ZooKeeper server with a ticktime of 2000 milliseconds to manage the heartbeats and session timeouts. The machine learning models are trained using scikit learn package in Python programming language for the Kafka-Storm data pipeline and Spark MLlib is used for the Kafka-Spark data pipeline which is written in Scala programming language. We have used jdk 1.8.051, Python 2.7.14 and Scala version 2.10.6 for our implementation.

4.2 Machine Learning

The data is prepared to predict the vitals in 10 seconds in the future. After the data preprocessing step, there were 9 million training samples, 1 million testing samples and 27 features. A Pearson correlation coefficient is calculated to see how the selected features are correlated to each other which shown in the figure below.

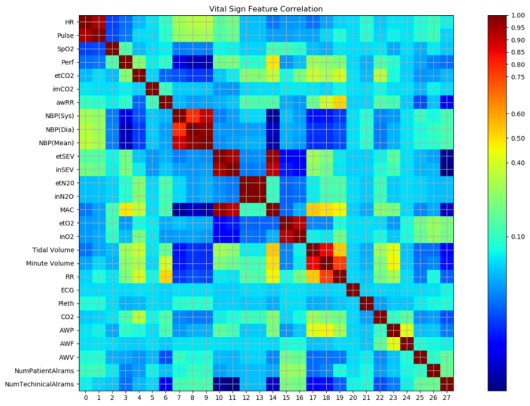

Figure 4.1: Pearson Correlation Coefficients of the data

We have found that some of the features like the heart rate and pulse rate are very highly correlated whereas the CO_2 concentration measurements have low correlation values with other features. This was the reason the accuracies of all the machine learning algorithms used in this experiment were in 30% for CO_2 predictions. We have used multiple linear regression, random forest regression, gradient boosting regression and multilayer perceptron regression algorithm to predict heart rate, pulse rate and CO_2 levels in this experiment. The values of accuracies and root

mean squared errors for all the models are presented in the table 4.1 and different plots are shown figure 4.2 and 4.3.

Table 4.1: Accuracies and Root Mean Squared Errors

	MLR(HR)	MLR(PR)	MLR(CO2)	RF(HR)	RF(PR)	RF(CO2)
RMSE	2.53	2.24	13.88	2.61	2.35	14.23
Accuracy	0.89	0.92	0.22	0.89	0.91	0.18

	GB(HR)	GB(PR)	GB(CO2)	MLP(HR)	MLP(PR)	MLP(CO2)
RMSE	2.54	2.26	12.53	2.59	2.91	12.53
Accuracy	0.89	0.92	0.36	0.8893	0.8631	0.3614

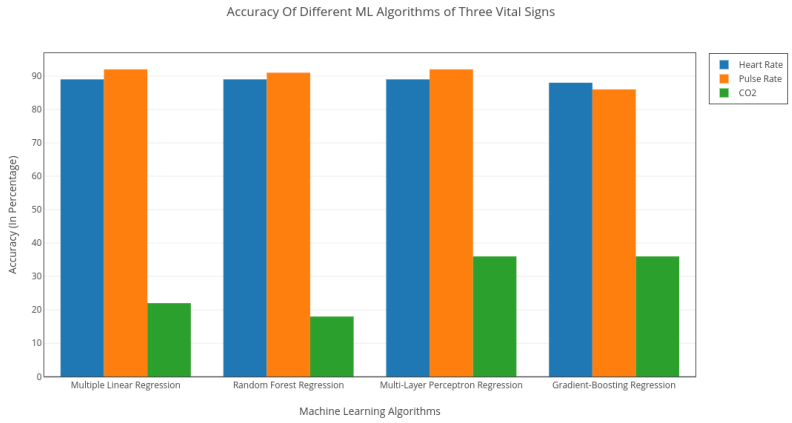

Figure 4.2: Accuracy of Machine Learning Models

4.3 Throughput and Latency of Storm

In this experimental setup, we conducted a simulation where a single user Kafka stream produced events continuously. Several configurations of parallelism were tested to collect the number of tuples per second and the time it takes to process the events. The throughput and latency are shown in below table and plots. The naming convention such as "8-10 Sp, 8-12 Bt, 8-10 Bt" in

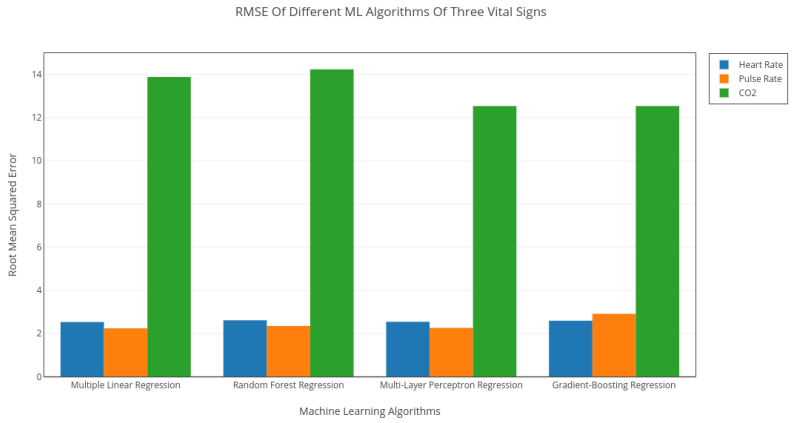

Figure 4.3: RMSE of Machine Learning Models

the parallelization configuration means that we have the "Kafka-Spout" running on 8 executors with 10 number of tasks, "SplitBolt" running on 8 executors with 12 tasks and "PredictionBolt" running on 8 executors with 10 number of tasks on each executor. We also collected results for the throughput and latency where the spouts and bolts were running on a single physical machine without any level of parallelism. In this experiment, we observed that the throughput increases with the increase in parallelism, but it stabilizes after a certain extent of concurrency. Similarly, the latency increases with the increase in parallelism and then it holds steady. In our case, we found that "8-10 Sp, 8-12 Bt, 8-10 Bt" configuration has the overall best performance.

Table 4.2: Throughput and Latency of Storm with Different Parallelization configurations

Parallelization Configuration	Throughput (Number Of Tuples/sec)	Latency (Time in ms)
No Parallelism	25219	2.454
2-4 Sp, 3-6 Bt, 2-4 Bt	27304	2.814
4-6 Sp, 8-10 Bt, 4-8 Bt	32002	2.318
8-10 Sp, 8-12 Bt, 8-10 Bt	32828	2.373

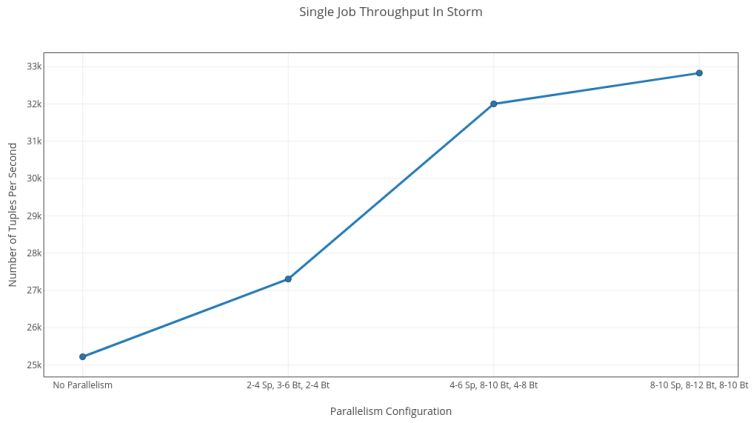

Figure 4.4: Throughput In Storm Different Parallelization configurations

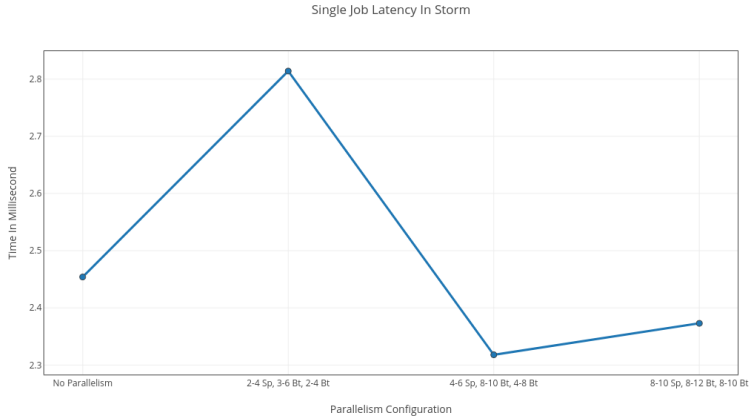

Figure 4.5: Latency In Storm Different Parallelization configurations

4.4 Throughput and Latency of Spark

In this benchmark, we compared the performance of Spark streaming using a single receiver and multiple receivers getting the input streams from a single Kafka topic. We implemented 5 DStreams for receiving data from a single user in different batch intervals. With increase in batch intervals, the streaming application starts queuing up the results in backlog which eventually brings down the streaming job. We observed that with the increase in the batch interval, the latency increases because of the increase in scheduling delays. We obtained higher throughput with parallelization configuration as compared to the configurations without parallelization and the highest throughput is obtained for a batch interval of 25. The latency was pretty much higher in case of parallelization as compared to the configuration without parallelization. The results are shown in the below tables 4.3-4.4 and figures 4.6-4.7.

Table 4.3: Throughput and Latency of Spark with Different Parallelization configurations

Batch Interval(sec)	Throughput(Parallelization)	Latency(Parallelization)
5	56546	14161
10	76185	13181
15	78728	12430
20	80154	12824
25	81112	13062
30	76546	13815
35	75852	13056
40	75357	12478

4.5 Kafka Producer Input Rate for multiple users

We measured the input rate of Kafka stream in terms of average events per second for multiple users. Each of the users are simulated as a single topic with 3 number of partitions and 2 level of replication. The Kafka servers ran on a 3 broker cluster on different physical machines where one of the nodes act as a leader and others act as followers. For each of the experiments, we ran

Table 4.4: Throughput and Latency of Spark without Parallelization configurations

Batch Interval(sec)	Throughput(w/o Parallelization)	Latency(w/oParallelization)
5	51092	2037
10	51853	2432
15	52841	4274
20	53021	5380
25	57684	4939
30	52418	5915
35	50841	6355
40	48632	5494

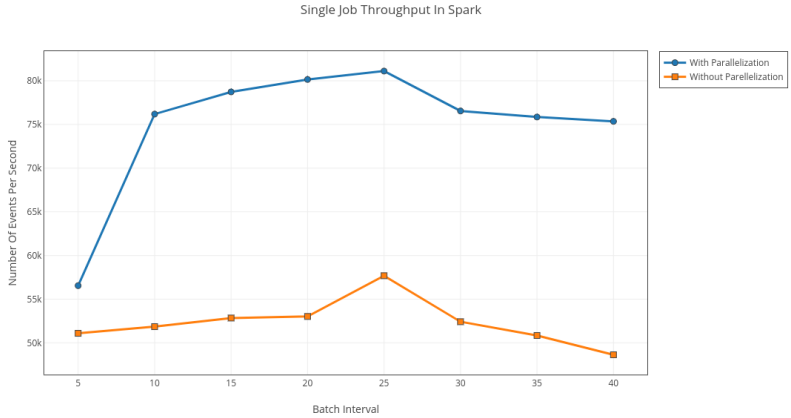

Figure 4.6: Throughput In Spark Different Parallelization configurations

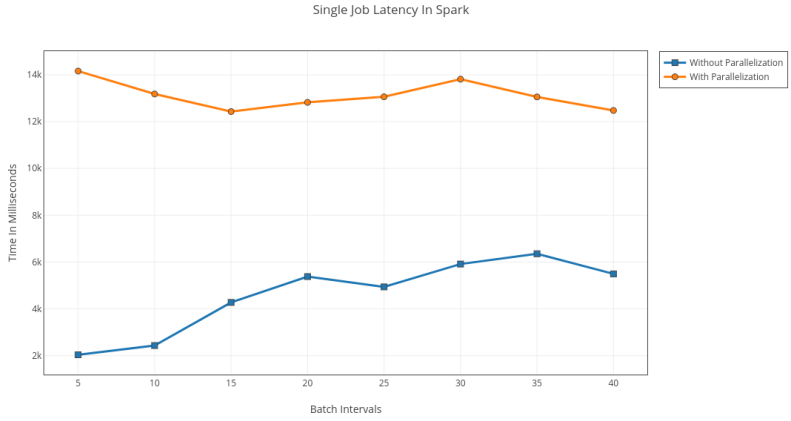

Figure 4.7: Latency In Spark Different Parallelization configurations

the simulation for nearly half hour in order to produce continuous streams of vital signs. It is observed that the number of events produced per second increased with the increase in the number of users. The pattern hold steady after a certain point. As per our use case, we found that maximum throughput is achieved for 40 users in our Kafka cluster.

Table 4.5: Kafka Producer Input Rate

Number Of Users	Kafka Producer Input Rate (avg. events/sec)
5	88154.63
10	88269.11
15	88463.19
20	88653.76
30	89176.58
40	90599.88
60	89541.12

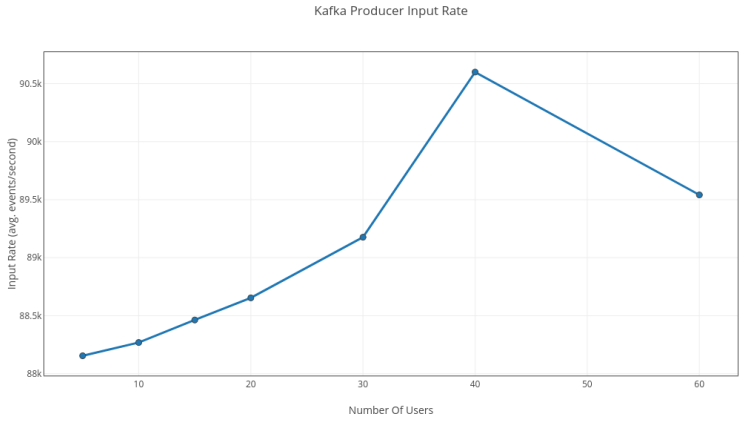

Figure 4.8: Kafka Producer Input Rate

4.6 Storm vs Spark Throughout and Latency Comparison

We conducted experiments for both Storm and Spark to compare the throughput and latency for multiple users being processed simultaneously. In Storm, we configured the Kafka spout to run

on 8 executors with 10 number of tasks on each of the executors. Similarly, the Split bolts were executed on 8 executors with 12 tasks, Predict bolts on 8 executors with 10 task and the result bolts on 8 executors with 10 tasks. In Spark, we received multiple user streams from Kafka in 10 DStreams which ran 5 different threads to receive data concurrently. All of these parallelisms have been introduced to achieve the best throughput and latency in an 18 node Storm and Spark cluster.

The throughput for Storm pipeline increased until 30 number of users and it became steady afterwards. However, in Spark the throughput was higher as compared to Storm and we achieved best throughput for nearly 40 number of users. Spark achieved better throughput by processing events in micro batches, but with the expense of huge latency. This is because of the scheduling delays in the parallelization configurations in Spark. However, Storm processed one record at a time for which it achieved low latency on an average of 19 milliseconds for 30 number of users. Hence, Spark performed better than Storm in terms of throughput, but Storm outperformed Spark in terms of latency for our use case.

The plots for the comparison of throughput and latency for multiple users in Storm and Spark are shown in the figures 4.9 and 4.10. The figure 4.11 shows the number of users supported for a single pipeline which has an upper bound of 35 for Storm and 45 for the Spark pipeline. The tables 4.6 and 4.7 includes the results collected for the throughput and latency values for Storm and Spark pipelines for multiple users.

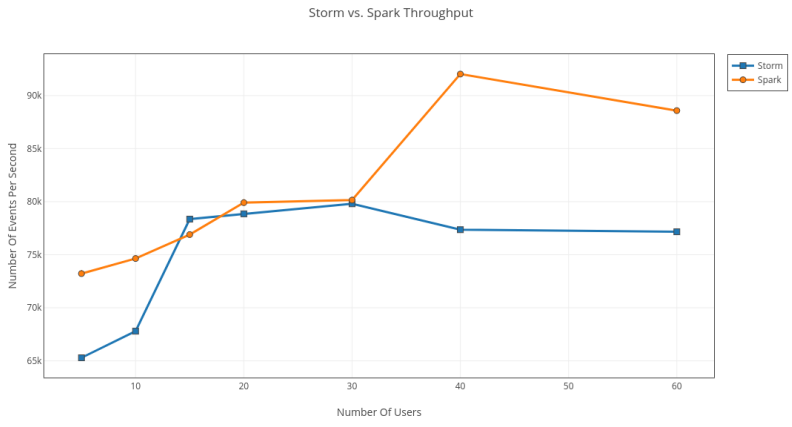

Figure 4.9: Storm vs Spark Throughput

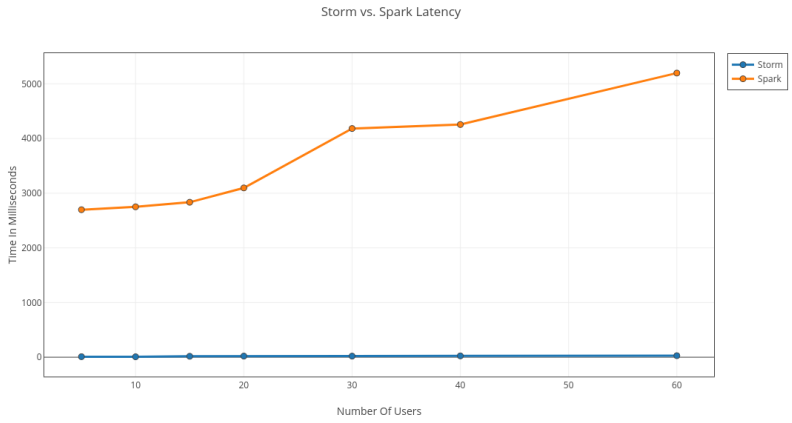

Figure 4.10: Storm vs Spark Latency

Table 4.6: Throughput of Storm vs Spark

Number of Users	Storm Throughput	Spark Throughput
5	65285	73220
10	67793	74641
15	78351	76895
20	78851	79904
30	79800	80145
40	77357	92021
60	77159	88578

Table 4.7: Latency of Storm vs Spark

Number of Users	Storm Latency	Spark Latency
5	9.342	2698
10	7.728	2751
15	16.852	2834
20	20.34	3096
30	19.91	4183
40	24.154	4256
60	27.632	5197

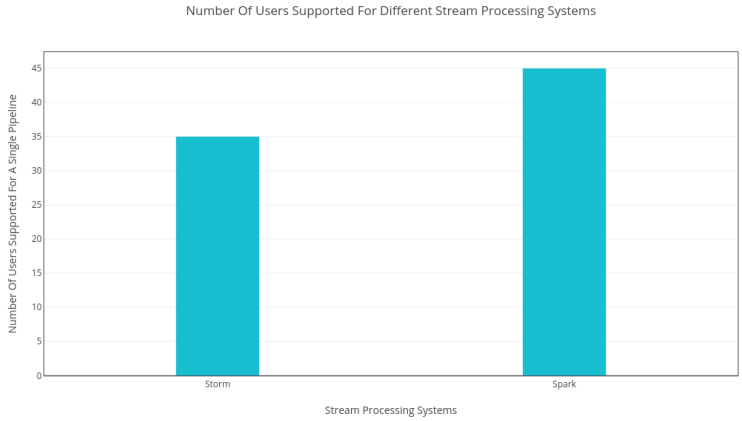

Figure 4.11: Number of Users Supported by Storm and Spark

Chapter 5

Related Work

This chapter is a synopsis of previous work done in this research area which provides an insight of the non-existence of important functionalities of the best possible methodologies. It has been agreed in the research community to improve upon the existing approaches which justifies the significance of the work.

It is predicted by the Cisco Internet Business Solutions Group (IBSG) that there will be 50 billion connected devices by 2020 supporting various applications such as healthcare services, air pollution monitoring, transportation, energy etc. These massive stream of small pieces of spatio-temporal and heterogeneous data needs to be processed in real time [7]. Health monitoring and e-Inclusion impose challenges to process data stream to be processed with near real-time constraints [19]. Real time processing of this massive flow of data do not scale to process billions or trillions of tuples in the traditional cluster based solution. The fully decentralized stream processing engines addresses the challenges of scalability, heterogeneity, incompleteness, timeliness and privacy in real time data processing [7]. The healthcare services can be improved by providing patient centric services, earlier detection of spreading diseases, monitoring the quality of the hospitals and by improving the treatment methods. The proposed big data healthcare architecture constitutes of a heterogeneous dataset as an input to HDFS through Flume and Sqoop where data analysis is performed using Map-Reduce and Hive. It implements machine learning algorithms to predict the risk of patient health condition at the earlier stages. Hbase is used to store multi-structured data and Apache Storm performs live streaming. AWS lambda functions are used to transmit emergency signals and reports are generated through intellicius and hunk [4]. The method of predictive analytics incorporates a variety of techniques from data mining, statistics and game theory that uses current and past data with statistical or other analytical model to predict events in future [3]. However, using the batch processing Hadoop map-reduce framework does not satisfy the key constraints of real time and stream data processing.

The occurrences of different health conditions can be regarded as the complex events. A system is developed using the public-subscribe system and cluster computing framework on top of health data. Multiple users are running Kafka producer clients to send data in real-time and Spark streaming is used to process data from Kafka of different window sizes to analyze the health conditions. Bayes prediction algorithm is used to predict congestive heart failure risk and also stress index is detected [20]. Chen et al. discussed the overview of data stream management system (DSMS) called T2 which is a framework that provides a programming infrastructure to run user-defined functions on streams, an suitable adapter layer to covert different streams into streams of java objects, thread management for query scheduling, SQL-like query language and time window constraints. Electrocardiogram (ECG) data streams are analyzed by monitoring the patient's heart rate and accelerometer readings using T2 [21], but predictive analytics is not performed and performance comparison of T2 is not conveyed in this study. Non-parametric Bayesian models are used by an algorithmic framework to process real-time physiological data collected by Sprout data aggregator. The wearable device periodically transmits data to the backend server to perform computationally intensive tasks and builds custom user profiles based on inferred hidden Markov states [5]. A static lattice model is proposed for information flow control over high volume and velocity of sensing data stream to build powerful data analytics systems with real time event detection for smart health. The static lattices are initialized for a source wearable sensor with three levels of data sensitivity and for users with three levels of data access and this comparison works just after security verification of data packets at Data Stream Management System (DSMS). The performance of the proposed mechanism is evaluated in a real-time Apache Kafka cluster [22].

A model-driven methodology with the combination of autonomic computing and cognitive computing design pattern is proposed for the design and development of smart IoT based systems. A smart monitoring system is developed in this context to manage patient's diabetes by measuring blood sugar level using wearable devices. The Big Data Stream Detection pattern is managed to detect issues using near real time processing and the Big Data Analytic Predictive pattern predicts new information using batch processing. Asynchronous data streams are received by Apache

Kafka which is consumed by Apache Storm to perform data normalization, data storage in distributed cluster for analysis and detection of problem at near real time using a topology of three bolts [1]. A platform based on Artemis cloud platforms with relevance in online health analytics for Neonatal Intensive care is presented to enable real time astronaut monitoring for prognostics and health management within space medicine using online health analytics. Artemis employs IBM's Infosphere Streams, a novel streaming middleware system that enables multi-stream temporal analysis in real time for clinical management and then enables data storage within the data persistency component. The physiological and clinical data are sent to platform component at mission control at each stage when the communication is available and researchers at the mission control deploy and tailor predictive analytics and diagnostics [23]. A real time intelligent perioperative system is proposed that assesses the risk of postoperative complications (PC) after aggregating and transforming EHR data from different patients and dynamically interacts with physicians to improve the predictive results. Apache Kafka distributed message queue, Apache Spark, Cassandra and HDFS are used as an input agent, distributed streaming computational infrastructure, NoSQL database and distributed file system solution respectively. The batch model training is established on the distributed machine learning tools of Apache Spark Mlib and TensorOnSpark for deep learning tasks [24].

Xhafa et al. evaluated Yahoo!S4 for Big Data Stream processing following the actor model for distributed computing and exemplified with data streams received from FlightRadar24 global flight monitoring system. They obtained geo information of international flights covered by ADS-B network supplemented with other analytical information such as departure/destination airport names, flight status, country closer to the current position etc. They also discussed issues of rate of data ingestion or generation, structured/unstructured data variety, low/high volume of data that can appear along the data streams. Nonetheless, window and chain based sampling techniques could be used to deal with incoming flow of data and to resolve memory issues while considering large windows [25]. Boeing Research and Technology Advanced Air Traffic Management built a system to perform predictive analytics over streams of big aviation data using IBM's InfoSphere, Web-

Sphere message broker for real time message brokering, other statistical analytical tools etc. [26]. A combined model of machine learning (ML) with complex event processing based on open source components is implemented to predict complex events for proactive IoT applications. An adaptive moving window regression algorithm is proposed for dynamic IoT data streams and the spectral components of time series data is exploited to find the optimum size for training window while the error propagates through the system. The AMWR representing the machine learning component accesses the real time traffic data provided by city of Madrid from the Apache Kafka topic. It published the event tuple into a different topic which is received by Esper complex event processing platform to perform pattern matching to produce complex events. However, the proposed model seems to be problematic while considering the error propagation through the system, choosing the optimal prediction window size and hyper-parameter tuning of machine learning models in case of online model training [27]. Sharif et al. and Rizwan et al. discussed the real time streaming events and predictive analytics in smart traffic system on a higher level without providing any details of actual methodologies [28] [29].

The data-driven applicability of streaming analytic system that enable companies to collect real time, heterogeneous plant data with steps of text extraction, causal correlation, statistical modelling, real time monitoring and anomaly detection is proposed to improve overall equipment effectiveness of semiconductor industrial manufacturing. The streaming analytics components for manufacturing performance monitoring and predictive adjustment involves the use Apache Storm, HDFS, Redis and interoperable HTTP API for data analysis based on R [2]. Hadoop Map-Reduce framework is used to perform data mining analytic operation and prediction model for movie review is built by using Naive Bayes Classifier [30]. Map-Reduce framework is used to analyze stream events such as carbon monoxide levels generated from sensors [31]. These are imperfect designs to choose a batch processing system to process stream events. An enhanced structural health monitoring system using stream processing and artificial neural network (SPANNeT) techniques has been developed to monitor bridge structure in real time and a warning system is deployed based upon measured bending strains [32]. A real time anomaly detection in streaming

heterogeneity (RADISH) system is built for rapidly detecting patterns and anomalies in streaming data which is composed of a streaming machine learning process and streaming anomaly detection process. The streaming feature vector generation and online modelling training components of RADISH-L are built using Apache's Spark streaming engine which load, update and save models and these models are retrieved by RADISH-A using ESPER open source complex event processing framework [33]. Kejela et al. chose Oxdata's H2O for monitoring drilling processes and equipment in an oil and gas company, after studying Mahout, RHadoop and Spark for fast in-memory processing, strong machine learning engine and accurate predictive analytics is used to estimate missing value and to replace incorrect reading [34]. The architecture of dynamic real-time modeling coined as adaptive modeling is implemented by a cascading technology using Apache Spark streaming and Apache Kafka. It is described based on upon automated models which adapt to real time customer behavior via model feedback loops. It is suggested that it works from the perspective of a simplified rendition of a demand-side platform player in a real time bidding setting for mobile advertising [35].

The Granules distributed stream processing engine [36] [37] was designed over a publish-subscribe infrastructure [38] [39] and incorporated support for MapReduce has been deployed in the context of processing electroencephalograms [40] [41]. The underlying publish/subscribe infrastructure underpinning Granules has been deployed in the context of real-time applications such as audio/video conferencing [42] and peer-to-peer grids [43]. The Neptune [44] stream processing engine is designed specifically for data generated in sensing environments such as IoT and Cyber Physical systems. Optimal stream scheduling is NP-Hard, and efforts have focused on the use of statistical and machine learning based algorithms to orchestrate scheduling. The prediction-ring algorithm [45] combines forecasting based on time-series models to make incremental migrations to preserve low-latency, high-throughput processing. The Hermes system [46] combines stream processing with federation across cloud and fog nodes to support integrated query processing. The Synopsis system combines real-time stream processing with compact data structures to dynamically construct sketches from spatiotemporal observations [47].

Frameworks for enabling data fitting have been explored in the context of epidemiology [48] [49], spatiotemporal phenomena [50] [51] [52], anomaly detection [53], and cloud deployments [54].

Efforts in distributed storage of data streams include the Galileo [55] [56] system for spatiotemporal data, and Gossamer for storage of data generated in continuous sensing environments.

Chapter 6

Conclusion and Future Work

This thesis demonstrates the methodologies to process physiological data in near real time using data pipelines. This work presented the performance comparison between the two dominant stream processing engines, i.e. Apache Storm and Spark which are used to build these pipelines.

As per our simulation results, the throughput and latency of Storm increases with the level of parallelism in the spouts and bolts and with the number of worker nodes in the cluster. Similarly, our experiments suggest that Spark performs well for a batch interval of 25 and at a certain level of concurrency for our use case when receiving inputs from a single source. These parameters become stable after a certain level which we chose as our best configuration to run all of our experiments. This thesis also demonstrates that any machine learning models can be used to perform online prediction of any vital signs in parallel.

Our three-broker Kafka system could support forty number of users with nearly 90 thousand average events per second. Spark achieved higher throughput up to 92 thousand events per second for 40 to 45 number of users, but Storm could only process a maximum of 80 thousand tuples per second for 30 to 35 number of users. However, Spark introduces a huge latency of 4 seconds to achieve the highest throughput where as Storm processes with a minimal average processing delay of 19 milliseconds. We suspect that the network bandwidth and the hardware configuration setup could be a bottleneck for Storm's throughput during the runtime of our experiments. Hence, we can conclude that Storm is super fast as compared to Spark streaming and achieves similar throughput for the physiological data processing.

Storm has provided us a solid benchmark to extend this research into future. Our future work will target to explore the use of different data fitting algorithms for generation of predictions such as Convolutional Neural Networks(CNNs). We are planning to explore how the data pipeline construction changes with variations in the type of mission critical applications. Some use cases may prioritize latency while others may prioritize throughput. We also plan to explore the use

of other stream processing engines such as Apache Flink, Apache Samza etc. to construct the processing pipelines.

Bibliography

[1] Emna Mezghani, Ernesto Exposito, and Khalil Drira. A model-driven methodology for the design of autonomic and cognitive iot-based systems: Application to healthcare. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):224–234, 2017.

[2] Yi-Hsin Wu, Sheng-De Wang, Li-Jung Chen, and Cheng-Juei Yu. Streaming analytics processing in manufacturing performance monitoring and prediction. In *Big Data (Big Data), 2017 IEEE International Conference on*, pages 3285–3289. IEEE, 2017.

[3] T Eswari, P Sampath, S Lavanya, et al. Predictive methodology for diabetic data analysis in big data. *Procedia Computer Science*, 50:203–208, 2015.

[4] J Archenaa and EA Mary Anita. A survey of big data analytics in healthcare and government. *Procedia Computer Science*, 50:408–413, 2015.

[5] Lawrence Chow and Nicholas Bambos. Real-time physiological stream processing for health monitoring services. In *e-Health Networking, Applications & Services (Healthcom), 2013 IEEE 15th International Conference on*, pages 611–616. IEEE, 2013.

[6] R Vanathi and A Shaik Abdul Khadir. A robust architectural framework for big data stream computing in personal healthcare real time analytics. In *Computing and Communication Technologies (WCCCT), 2017 World Congress on*, pages 97–104. IEEE, 2017.

[7] Rudyar Cortés, Xavier Bonnaire, Olivier Marin, and Pierre Sens. Stream processing of healthcare sensor data: studying user traces to identify challenges from a big data perspective. *Procedia Computer Science*, 52:1004–1009, 2015.

[8] David Liu, Matthias Görges, and Simon A Jenkins. University of queensland vital signs dataset: Development of an accessible repository of anesthesia patient monitoring data for research. *Anesthesia & Analgesia*, 114(3):584–589, 2012.

[9] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[10] Van-Dai Ta, Chuan-Ming Liu, and Goodwill Wandile Nkabinde. Big data stream computing in healthcare real-time analytics. In *Cloud Computing and Big Data Analysis (ICCCBDA), 2016 IEEE International Conference on*, pages 37–42. IEEE, 2016.

[11] Apache Zookeeper. Apache software foundation. http://zookeeper.apache.org/, accessed June 2018.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[13] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[14] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[15] Spark Streaming Programming Guide. Spark streaming. https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html, accessed June 2018.

[16] Apache Kafka Tutorial. Apache kafka. https://www.tutorialspoint.com/apache_kafka/index.htm, accessed June 2018.

[17] Apache Kafka. Apache kafka1. https://kafka.apache.org/, accessed June 2018.

[18] ZooKeeper Overview. Apache zookeeper. https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription, accessed June 2018.

[19] Heiko Schuldt and Gert Brettlecker. Sensor data stream processing in health monitoring. 2003.

[20] Sandeep Singh Sandha, Mohammad Kachuee, and Sajad Darabi. Complex event processing of health data in real-time to predict heart failure risk and stress. *arXiv preprint arXiv:1707.04364*, 2017.

[21] Chung-Min Chen, Hira Agrawal, Munir Cochinwala, and David Rosenbluth. Stream query processing for healthcare bio-sensor applications. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 791–794. IEEE, 2004.

[22] Deepak Puthal. Lattice-modelled information flow control of big sensing data streams for smart health application. *IEEE Internet of Things Journal*, 2018.

[23] Carolyn McGregor. A platform for real-time online health analytics during spaceflight. In *Aerospace Conference, 2013 IEEE*, pages 1–8. IEEE, 2013.

[24] Zheng Feng, Rajendra Rana Bhat, Xiaoyong Yuan, Daniel Freeman, Tezcan Baslanti, Azra Bihorac, and Xiaolin Li. Intelligent perioperative system: Towards real-time big data analytics in surgery risk assessment. *arXiv preprint arXiv:1709.10192*, 2017.

[25] Fatos Xhafa, Victor Naranjo, and Santi Caballé. Processing and analytics of big data streams with yahoo! s4. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, pages 263–270. IEEE, 2015.

[26] Samet Ayhan, Johnathan Pesce, Paul Comitz, David Sweet, Steve Bliesner, and Gary Gerberick. Predictive analytics with aviation big data. In *Integrated Communications, Navigation and Surveillance Conference (ICNS), 2013*, pages 1–13. IEEE, 2013.

[27] Adnan Akbar, Abdullah Khan, Francois Carrez, and Klaus Moessner. Predictive analytics for complex iot data streams. *IEEE Internet of Things Journal*, 4(5):1571–1582, 2017.

[28] Abida Sharif, Jianping Li, Mudassir Khalil, Rajesh Kumar, Muhammad Irfan Sharif, and Atiqa Sharif. Internet of thingsâĂŤsmart traffic management system for smart cities using big data analytics. In *Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2017 14th International Computer Conference on*, pages 281–284. IEEE, 2017.

[29] Patan Rizwan, K Suresh, and M Rajasekhara Babu. Real-time smart traffic management system for smart cities by using internet of things and big data. In *Emerging Technological Trends (ICETT), International Conference on*, pages 1–7. IEEE, 2016.

[30] DN Disha, BJ Sowmya, S Seema, et al. An efficient framework of data mining and its analytics on massive streams of big data repositories. In *Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), IEEE*, pages 195–200. IEEE, 2016.

[31] Marco Aurelio Borges, Paulo Batista Lopes, Leandro A Silva, Massaki de O Igarashi, and Gabriel Melo F Correia. An architecture for the internet of things and the use of big data techniques in the analysis of carbon monoxide. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 184–191. IEEE, 2017.

[32] Ittipong Khemapech. Bridge structural monitoring and warning system application in thailandâĂŤexperiences learned. In *TRON Symposium (TRONSHOW), 2017*, pages 1–8. IEEE, 2017.

[33] Brock Böse, Bhargav Avasarala, Srikanta Tirthapura, Yung-Yu Chung, and Donald Steiner. Detecting insider threats using radish: a system for real-time anomaly detection in heterogeneous data streams. *IEEE Systems Journal*, 11(2):471–482, 2017.

[34] Girma Kejela, Rui Maximo Esteves, and Chunming Rong. Predictive analytics of sensor data using distributed machine learning techniques. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 626–631. IEEE, 2014.

[35] Donald Kridel, Daniel Dolk, and David Castillo. Adaptive modeling for real time analytics: The case of" big data" in mobile advertising. In *System Sciences (HICSS), 2015 48th Hawaii International Conference on*, pages 887–896. IEEE, 2015.

[36] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. An overview of the granules runtime for cloud computing. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 412–413. IEEE, 2008.

[37] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[38] Geoffrey Fox and Shrideep Pallickara. Deploying the naradabrokering substrate in aiding efficient web and grid service interactions. *Proceedings of the IEEE*, 93(3):564–577, 2005.

[39] Geoffrey Fox, Sang Lim, Shrideep Pallickara, and Marlon Pierce. Message-based cellular peer-to-peer grids: foundations for secure federation and autonomic services. *Future Generation Computer Systems*, 21(3):401–415, 2005.

[40] Kathleen Ericson, Shrideep Pallickara, and Charles W Anderson. Analyzing electroencephalograms using cloud computing techniques. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 185–192. IEEE, 2010.

[41] Kathleen Ericson, Shrideep Pallickara, and Charles W Anderson. Failure-resilient real-time processing of health streams. *Concurrency and Computation: Practice and Experience*, 27(7):1695–1717, 2015.

[42] Ahmet Uyar, Shrideep Pallickara, and Geoffrey C Fox. Towards an architecture for audio/video conferencing in distributed brokering systems. In *Communications in Computing*, pages 17–23, 2003.

[43] Geoffrey Fox, Sang Lim, Shrideep Pallickara, and Marlon Pierce. Message-based cellular peer-to-peer grids: foundations for secure federation and autonomic services. *Future Generation Computer Systems*, 21(3):401–415, 2005.

[44] Thilina Buddhika and Shrideep Pallickara. Neptune: Real time stream processing for internet of things and sensing environments. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 1143–1152. IEEE, 2016.

[45] Thilina Buddhika, Ryan Stern, Kira Lindburg, Kathleen Ericson, and Shrideep Pallickara. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569, 2017.

[46] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Hermes: Federating fog and cloud domains to support query evaluations in continuous sensing environments. *IEEE Cloud Computing*, 4(2):54–62, 2017.

[47] Thilina Buddhika, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Synopsis: A distributed sketch over voluminous spatiotemporal observational streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2552–2566, 2017.

[48] Matthew Malensek, Walid Budgaga, Sangmi Pallickara, Neil Harvey, F Jay Breidt, and Shrideep Pallickara. Using distributed analytics to enable real-time exploration of discrete event simulations. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 49–58. IEEE Computer Society, 2014.

[49] Walid Budgaga, Matthew Malensek, Sangmi Pallickara, Neil Harvey, F Jay Breidt, and Shrideep Pallickara. Predictive analytics using statistical, learning, and ensemble methods to support real-time exploration of discrete event simulations. *Future Generation Computer Systems*, 56:360–374, 2016.

[50] Matthew Malensek, Walid Budgaga, Ryan Stern, Shrideep Pallickara, and Sangmi Pallickara. Trident: Distributed storage, analysis, and exploration of multidimensional phenomena. *IEEE Transactions on Big Data*, 2018.

[51] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Analytic queries over geospatial time-series data using distributed hash tables. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1408–1422, 2016.

[52] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Fast, ad hoc query evaluations over multidimensional geospatial datasets. *IEEE Transactions on Cloud Computing*, 5(1):28–42, 2017.

[53] Walid Budgaga, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. A framework for scalable real-time anomaly detection over voluminous, geospatial data streams. *Concurrency and Computation: Practice and Experience*, 29(12), 2017.

[54] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas. Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 73–80. IEEE Computer Society, 2012.

[55] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Galileo: A framework for distributed storage of high-throughput data streams. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 17–24. IEEE, 2011.

[56] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Polygon-based query evaluation over geospatial data using distributed hash tables. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 219–226. IEEE Computer Society, 2013.