Date of acceptance Grade

Instructor

Distributed approach to analyze physiological time series signals in medical telemetry

Maninder Pal Singh

Helsinki May 12, 2016

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Maninder Pal Singh

Työn nimi — Arbetets titel — Title

Distributed approach to analyze physiological time series signals in medical telemetry

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	May 12, 2016	74 pages + 13 appendices

Tiivistelmä — Referat — Abstract

Research in healthcare domain is primarily focused on diseases based on the physiological changes of an individual. Physiological changes are often linked to multiple streams originated from different biological systems of a person. The streams from various biological systems together form attributes for evaluation of symptoms or diseases. The interconnected nature of different biological systems encourages the use of an aggregated approach to understand symptoms and predict diseases. These streams or physiological signals obtained from healthcare systems contribute to a vast amount of vital information in healthcare data. The advent of technologies allows to capture physiological signals over the period, but most of the data acquired from patients are observed momentarily or remains underutilized. The continuous nature of physiological signals demands context aware real-time analysis. The research aspects are addressed in this thesis using large-scale data processing solution.

We have developed a general-purpose distributed pipeline for cumulative analysis of physiological signals in medical telemetry. The pipeline is built on the top of a framework which performs computation on a cluster in a distributed environment. The emphasis is given to the creation of a unified pipeline for processing streaming and non-streaming physiological time series signals. The pipeline provides fault-tolerance guarantees for the processing of signals and scalable to multiple cluster nodes. Besides, the pipeline enables indexing of physiological time series signals and provides visualization of real-time and archived time series signals. The pipeline provides interfaces to allow physicians or researchers to use distributed computing for low-latency and high-throughput signals analysis in medical telemetry.

ACM Computing Classification System (CCS):
Information systems → Information systems applications → Computing platforms

Avainsanat — Nyckelord — Keywords
layout, summary, list of references

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

The digital technology is becoming more pervasive and it is transforming our lives. The constantly changing environment generates data in the form of time series to sustain relevant and valuable results. This is evident from the growth of massive data domains such as trading, gaming, advertisements, social networks, meteorology and medicine. Namely, various medical examinations rely on time series signals to measure health parameters of an individual. These health parameters can be obtained in the form of time series signals from electroencephalogram (EEG), electrocardiogram (ECG), electromyography (EMG) and other instruments.

The major part of the time series signals, such as EEG signals, are involuntary in nature, which makes them effective for brain fingerprinting [48]. Hence, they can also be used in criminal investigation [62], where EEG acts as biological fingerprints of an individual. In addition, EEG is considered as an identifier of a brain, which includes encoded information. This encoded information within a brain can help to identify individuals [76], locate mobility traces [53], object identification [75], locating sensitive information, identification of depression patterns [54], linguistic learning patterns [70], and diagnosis of various diseases [39]. Therefore, the analysis of such signals allows to unlock the potential of the brain and further contribute towards brain research. Considering this, the context of these time series signals and portability of the machines used to gather them evolve interest of many researchers to analyze the dynamics of these time series signals [35, 59]. Emotiv EEG headset [9], Mitsar Portable EEG System [15] are some examples of such portable devices.

The time series signals in medical telemetry are continuous in nature and manifest large scale data over the period. The advent of technologies allows to capture physiological signals over the period, but most of the data acquired from patients are momentarily observed or underutilized [79]. The physiological time series signals are often linked to multiple streams and originate from different biological systems of an individual. Considering the interconnectedness of various biological systems, the aggregated approach is more appropriate for prognosis or diagnosis of an individual. In order to have a comprehensive physiological context of time series signals, we need to develop new approaches to analyze physiological signals in medical telemetry. There have been attempts to process continuous physiological signals to provide better healthcare [33, 34, 52, 87]. The continuous nature of physiological signals demands context aware real-time analysis. Such

analysis demands the system to efficiently process and analyze these time series signals within determined time frame to derive valuable conclusions for diagnosis or prognosis. The existing systems are limited in scope and does not provide context aware real-time analysis of physiological time series signals originated from interconnected biological systems in an aggregate and cumulative manner.

The limitations of existing systems provided an opportunity to research the distributed approach to analyze physiological time series signals in medical telemetry. The research contributed towards a unified coherent system for processing and analyzing these signals in distributed computing environment. The research also explored existing state-of-the-art systems and, identified Spark and Spark Streaming as a possible environment to create a general purpose distributed data processing pipeline for analysis of time series signals. The focus has been given to the creation of a unified pipeline for processing of streaming, and non-streaming time series signals originated from homogeneous or heterogeneous data sources. An exhaustive discussion has been performed to sketch a solution. The proposed pipeline and provided interfaces enable researchers to use distributed computing for low-latency and high-throughput signals analysis in medical telemetry. It also ensures scalability and extensibility as per domain context. In addition, the sources such as PhysioNet [17] contain diverse collections of physiological signals contributed by different researches. These diverse collections are underutilized due to the lack of tools, which can perform cumulative processing of large scale time series data. The proposed system provides possibilities to use the collections of physiological signals for research and experimentation.

The research involved in this thesis explored various aspects linked to physiological time series signals. It includes identification of the existing systems for the analysis of physiological signal in medical telemetry and exploration of existing systems in improving healthcare for individuals. It also includes discussion of approaches used in time series signals analysis in distributed computing environment and investigation of existing open-source large-scale data processing systems for analyzing time series signals. In addition, the research involves a study of the possibilities to provide a general purpose distributed pipeline which can receive homogeneous or heterogeneous data sources and decouple them from data processing engine to provide on the fly management of data streams in the system. Furthermore, the research provides an interface and algorithms to perform cumulative analysis of physiological signals. We made an attempt to provide a detailed discussion on the research aspects linked to the thesis.

The thesis is structured as follows. Chapter 2 introduces the theoretical background concepts used in this thesis. Section 2.1 provides information about medical telemetry and various physiological signals used in measurements of vital information. It also explores existing tools and frameworks employed in the research domain. Section 2.2 illustrates distributed data processing techniques and open source software components available for large-scale data processing. Section 2.3 provides an overview to Berkeley Data Analytics Stack and explore software elements of the stack, namely, Spark and Spark Streaming. Section 2.4 contains information about data analysis and visualization tools used in the pipeline. Chapter 3 describes the research methodology employed in this thesis for research and prototyping of the solution.

Chapter 4 presents our contribution to the research work. Section 4.1 introduces the design challenges encountered while conceptualization of the distributed analysis of physiological signals (DAPS) system. Section 4.2 explains data analysis workflow which includes data collection and pre-processing methodologies. Section 4.3 explores platform concepts involved in distributed analysis. It includes software components, system architecture and different data injection techniques. Section 4.4 introduces the application programming interfaces of the DAPS system. It includes various application examples. Section 4.5 concludes the chapter with the evaluation of the DAPS system. Chapter 4 discusses scientific contributions of the research work, limitations, practical impact and plans to improve the research work. Chapter 5 concludes the thesis.

2 Background

2.1 Medical Telemetry

Physiology is a branch of biology which deals with the scientific study of activities and functioning of living systems [67]. A living system, in turn, consists of different physiological processes, such as cardiovascular activities, which generate a particular type of physiological signals. As per Bertrand [45], a signal is a time varying function, which consists of one or more variables and include useful information. In a biological context, physiological signals contain vital information such as state or behavior of a biological system. For example, voltage potential measurement using a single or multiple electrodes placed on the chest or scalp of an individual is accounted as physiological signals. Physiological signals can depend on a single variable or multiple variables. Based on their dependence, these signals are categorized into one-dimensional or multidimensional [1].

Medical telemetry involves remote measurement of physiological signals from an individual for diagnosis or observation [50]. Medical telemetry is also known as biotelemetry. The physiological signals related to potential or current refer to bioelectric signals. Bioelectric signals are recorded using electrodes, which detect a difference in electric potential in a particular body part. Electrocardiogram (ECG), Electroencephalogram(EEG), Electromyogram (EMG), Electrooculogram (EOG), are examples of commonly used methods to measure the bioelectric signals [66]. Each of these signals measures the electric potential difference in a particular body part of a living system. For example, ECG is used for heart, EEG is used for brain, EMG is used for muscles, and EOG is used for functioning of eye muscles. For the purpose of thesis and simplicity, we are focused on non-invasive ECG and EEG measurements.

ECG is a non-invasive electrophysiological monitoring methodology to record the electrical activity of heart. ECG is measured by placing metal electrodes over the chest of an individual. Electrodes measure rhythmic electric potential changes when a heartbeat happens in the heart. Usually, 12 electrodes are used to record electric potential difference over the skin surface during a period. ECG measurement is a repeated process of different waves: P wave, QRS complex, and T wave as shown in Figure 1. These waveforms are formed in a continuous process of depolarization and repolarization of heart chambers. Depolarization and repolarization are coordinated activities in heart chambers linked to contraction and expansion of the heart.

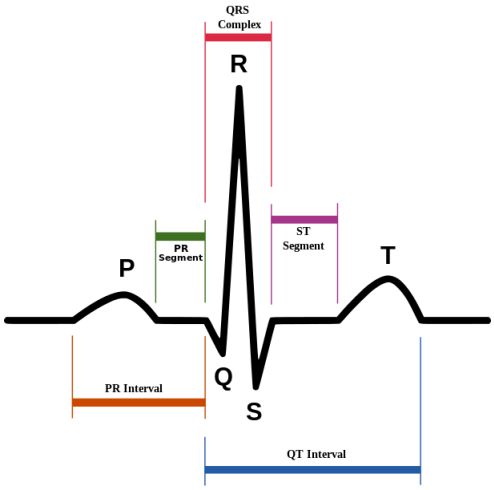


Figure 1: Schematic representation of normal sinus rhythm in ECG [43].

EEG is a non-invasive electrophysiological monitoring methodology to observe or record electric signals between brain cells [65]. EEG is measured by placing metal electrodes over scalp or cortical surface. The events or activities of an individual activate brain cells, and multiple electrodes placed on the scalp to measure the electric potential difference between brain cells or neurons. In a clinical context, the multiple electrodes are used to record the brain patterns of an individual. The brain patterns consist of brain waves, which are grouped into four broad categories based on their frequency bands: alpha (8-13 Hz), beta (greater than 13 Hz) , theta (4-8 Hz) and delta (0.5-4 Hz) [81]. These frequency bands or their combinations are associated with specific functions of a physiological process in a living system. EEG is helpful in the investigation of neurological disorders such as sleep disorders [40], epilepsy [77], and encephalopathy. Figure 2 presents an example of recorded EEG waves with various frequency bands.

Biological signals collected from a multitude of devices play a significant role in the prognosis of health disorders or clinical diagnosis of patients. Based on biological context, these signals either contain single-channel information or multi-channel information. A channel can include information from single electrode or combination of multiple electrodes. A greater number of electrodes used in the measurement of bioelectric signals provides a better quality and more descriptive information linked to a physiological process of a living system.

When these bioelectric signals are measured over a period, a time series is generated. The generated time series is also known as a physiological signal. For example, ECG is recorded to discover changes in electrical activities of the heart. Similarly, EEG is

Figure 2: Example of recorded EEG waves [42].

recorded to discover or diagnose neurological disorders of a person. In general, time series is defined as ordered sequence of successive values measured over a period. Time series can be one-dimensional or multi-dimensional. Bioelectric signals are mostly one-dimensional time series that are continuous in nature and non-stationary.

A collection of time series data assists in understanding of characteristics and patterns inscribed within a physiological process of a biological system. Biological signal processing mechanisms extract features from these time series and derives learning models around them. The later time series signals are mapped to previously generated learning models to derive conclusions and produce actionable insight. The statistics from the evaluation can provide better healthcare to individuals. Many other real world applications are actively using time series in various domains such as stock market, sales, advertisements, inventory analysis, and economics [36]. The time series found in different domains, other than medical telemetry, are beyond the scope of this thesis.

2.1.1 Traditional Data Analysis Workflow

The study of these physiological time series signals provides vital information linked to a physiological process, which helps in heart-attack prediction or identification of neurological disorders. It is a complex process to analyze nonlinear and non-stationary attributes of these physiological time series signals [61] [41]. Historically, the analysis of physiological time series signals involves four steps: data acquisition,

data pre-processing, feature extraction, and actionable insight. Figure 3 presents typical steps involved in biological signal processing.

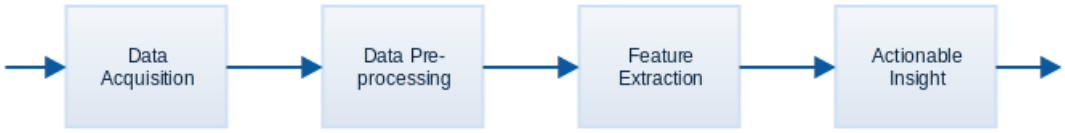


Figure 3: Block diagram of steps used in traditional analysis.

Data Acquisition. Historically, continuous time series signals are not stored in the medical devices. Previously, there have been endeavors to provide an interface to download data streams from medical devices. Due to the absence of protocols and standards, the interfaces vary with device manufacturers and are limited to proprietary software. Recent growth in electronics, medical device manufacturers are adopting common protocols to extract continuous time series signals. The emerging technologies provide sophisticated tools to record time series signals over a period. The continuous nature of physiological time series signal produces a large amount of data. Consequently, the volume of the data acquired at a velocity poses an explosion of data over a period. Therefore, most of the data acquired from patients are observed momentarily or remains underutilized. The momentarily observed data may not reflect the broad aspect of the clinical state of an individual patient. The coupling of physiological processes of the biological system requires long-term observation and data acquisition to deduce comprehensive perspective of an individual. Considering the data standards, data protocols, data privacy, and cost of medical devices, the continuous time series signal analysis require system that should be in coherent with existing healthcare infrastructure and provide extensibility for the future demands.

Data Pre-processing. Considering the disparity of medical devices, it is a daunting task to integrate continuous time series signals acquired from different sources. The signal streams need to organize and standardize into a common data formats that can be consumed by different components of a system. Sometimes, acquired time series signals contain redundant information or randomness, which is considered as noise in signal processing. For this reason, the signals are pre-processed by applying filters to eliminate noise from physiological signals. The pre-processing of signals leads to consistency in data that assists in the processing of signals at the next step. Due to privacy concerns and being a source of vital

information, proper handling of physiological time series signals is an important aspect of the system.

Feature Extraction. The pre-processed signals are complex in nature and linked to the physiological process of a biological system. The characteristics of signals are observed to extract biological features linked to the biological system. For example, feature extraction may involve identifying the difference in ECG pattern of an individual from regular ECG pattern in the case of heart related disorder. The observed patterns are used in clinical assessments in the next step.

Actionable Insight. The biological signal processing mechanism used to extract biological features are employed to create learning models. The learning models are then used to generate actionable insights. They are later used in clinical assessments. The data analysis process deduces pathological symptoms and identifies disorders linked to physiological time series signal obtained from an individual. For example, a heartbeat rhythm which is irregular, too fast or too slow, than the rhythm obtained from the learning model, may show a sign of arrhythmia [16].

2.1.2 Conventional Software Platforms

These four steps process acts as a general process for the development of various software platforms to process and analyze physiological signals. There are several platforms or frameworks that are used for analysis of biomedical signals. Most of these softwares are collection of modules for ECG or EEG. EEG modules assist in brain-computer interface (BCI) research and development. In brain research, BCI focuses on identifying user generated commands using brainwaves and performing an action related to commands in virtual or real world. BCI2000 [74], OpenViBE [71], BCILAB [56], EEGLAB [46], MIDAS [10] are some examples of software platforms that are used in the biomedical signal analysis.

BCI2000 is a general-purpose software platform designed in C++ programming language to use in BCI research. It provides application programming interface to extend modules for various building blocks such as data acquisition, signal processing, or user application [74].

OpenViBE is an open-source platform for design and development of BCI. It consists of easy to configure software modules to design BCI applications. It also supports integration of BCI with virtual reality applications. In addition, it supports graphical user interface (GUI) for designing BCI without using any

programming language.

BCILAB is a cross-platform open-source platform designed for development of brain-computer interface (BCI). It is a MATLAB-based platform designed to develop and test new methods for BCI [18]. It also provides a collection of existing common methods for fast pace development of BCI.

EEGLAB is a cross-platform open-source MATLAB toolbox, which provides an interactive graphic user interface for processing physiological signals such as EEG, magnetoencephalography (MEG) and electrocorticography (ECOG) data. It provides various functions such as pre-processing, visualization, and independent component analysis [19].

BioStream is a system for continuous monitoring of physiological signals [37]. Bar-Or et. al. [37] presented the prototype for monitoring and diagnosis of ECG. It uses Aurora [32] for data stream management and allows to create a patient plan using operators. It integrates alert mechanism to broadcast events to healthcare staff. For instance, an alert can be related to pathological symptoms linked to a patient heart.

MIDAS is a general-purpose distributed analysis system for physiological signals. It involves analysis of time series in a streaming environment. It is a Python based system, which vouches for modularity and easy to integrate system that can also be used for machine learning frameworks.

Software platforms, such as BCILAB and EEGLAB are MATLAB based software stacks. However, MATLAB is a proprietary software which is non-free. BCI2000, OpenViBE, BCILAB, and EEGLAB softwares are designed to target and collect data from a single user. Most of these softwares follow the traditional approach of handling biosignals sequential and configuring software stack based on GUI parameters to analyze the physiological signals. Some software stacks such as BCILAB provide limited applications in Human-Computer Interaction (HCI), whereas OpenViBE provides an interface to create applications in virtual reality. Systems such as BioStream are unreliable, because use of such a system in a large environment such as healthcare environment with many patients may cause alarm fatigue to the medical staff. Moreover, BioStream does not incorporate comprehensive context of physiological signals of patients. MIDAS is an attempt to analyze the physiologic signals in a distributed environment but limited to the degree of distribution. MIDAS also provides general purpose integration with Internet of Things (IoT). It is a naive system that is still under development.

Therefore, it is hard to rely on MIDAS for scalability and fault tolerance of ingested data. Most of the software stacks are too specific, e.g. BCI based research, which target signals from an individual. Moreover, some of them are too generic and limited in their context to process biomedical signals. Most of the research in the biomedical signal analysis are oriented towards individual signal processing and are non-distributed in nature. Physiological time series signals originated from a biological system are interconnected within the body. The time series signals require to be analyzed by embedding context awareness to the processing system. The system needs to have an open view to accept different modalities of data for comprehensive view of the physiological condition of a patient. It is more appropriate to use aggregated approach for processing and analysing these signals. Therefore, there is a need to develop a system that will utilize a continuous stream of real-time physiologic signals and record the signals to provide comprehensive clinical assessments.

Multiple data streams of physiological signals acquired in real-time require a capable system to process streams in stipulated time frame. In addition, a large environment such as hospitals with many patients may contribute to a multitude of physiological signals originated from biological systems of many patients. To provide a comprehensive viewpoint similar to the existing system, the streams from different sources should be distributed to perform parallel computation within a cluster of machines. Therefore, the computation is divided within a set of nodes and data streams are processed in a distributed environment. Considering the number and the rate of data streams, the cumulative data analysis requires a distributed data processing system to analyze physiological signals.

2.2 Distributed Data Processing

The data processing depends on the context of the domain and is also linked to the underlying data available in the area. The research and development in various fields lead to the evolution of emerging tools and methods to acquire data from different sources. For example, there has been consistent growth in the medical telemetry that leads to usage of technology for monitoring tools and softwares in medical domain. Another example is neuro telemetry, which employed EEG to monitor patients remotely. This method assists in the proactive monitoring of physiological symptoms of patients.

Traditional data processing methods involve one-to-one mapping of healthcare

systems to the patients. Healthcare setup may include more than one machines dedicated to a single patient for monitoring perspective. Such setup requires observers to monitor regularly the healthcare systems in a confined locality of the patient room. However, this setup is viable for small environments but limited in the context of computing and confined locality. Due to one-to-one mapping, the configuration requires lots of resources in large organizations.

The large organizations such as hospitals with many patients or agile research environment outlined the limitation of traditional data processing systems. In order to address the limitations, a proactive and aggregated approach is needed to handle the data streams from multitude of healthcare devices. This approach would make the decision making in a large setup dynamic. However, it requires a new computing paradigm. This emerging solution in the form of distributed systems appeared as an optimistic solution.

Distributed systems involve a collection of computation nodes connected via network, which share the resources and coordinate the computation between nodes. The structured organization of a multiple nodes provides additional computing power and allows flexibility to add or remove nodes as per requirement. Distributed systems collectively handle large-scale data domains and provides computation environment for data received from multiple resources.

The data processing in distributed systems can be categorized as batch processing and stream processing. In batch processing, the jobs or tasks are accumulated in batches over a period, and each batch is processed as a single unit. The scale, diversity, and volume of data governs the computing paradigm of distributed data processing. Evolution of the Internet, social networks, high-frequency trading, and large-scale systems adhere to have a real-time data processing requirements along with existing batch processing systems. The emergence of applications to handle incoming stream of real-time data at high rate leads to the evolution of distributed stream processing concept. The stream processing concept has evolved from stream computing paradigm, which involves continuous query and real-time analysis of stream data. The scalability, low latency, robustness, and fault-tolerance are general properties of the distributed stream processing solutions. There are various open-source and proprietary solutions or systems that address the requirements for distributed real-time stream processing such as Spark streaming, Amazon Kinesis, IBM Infosphere stream. The major part of our discussion is focused towards the open-source data processing solutions.

Table 1: Open-Source distributed data processing solutions

Solution	Type	Developed By
S4	Streaming	Yahoo
Storm	Streaming	BackType & Twitter
Spark	Batch & Streaming	AMPLab, UC Berkeley

Open-Source Distributed Data Processing Solutions. There are couple of solutions that are available for distributed data processing. Table 1 refers to various open-source solutions available for real-time large-scale data processing.

Storm is a distributed stream processing framework, developed in Clojure and built upon a model of task parallel computation [7]. It provides an adapter to write applications in virtually any language. Storm is optimized for low-latency processing and uses ZeroMQ[1] for message passing, which makes its architecture provide a guaranteed message processing [7]. It attempts to process each record at least once, and if a record is not yet processed by a node, it replays the records. In addition, it provides reliable fault detection and process management. On discovery of failure of a task, messages are automatically re-assigned by quickly restart the processing. For optimal resource handling, the processes in Storm are managed by supervisors.

S4 (Simple Scalable Streaming System) is a general purpose distributed and scalable streaming platform that allows the processing of continuous unbounded streams of data. Its processing model is inspired by MapReduce and uses key based programming model [63]. The computation is performed by processing elements, and messages are transmitted to them in the form of data events [64].

Spark is an in-memory distributed data analysis platform and provides Spark Streaming as an extension of the core Spark application programming interface (API) [21]. Spark is built upon the model of data parallel computation. It provides reliable processing of live streaming data. Spark streaming transforms streaming computation into a series of deterministic micro-batch computations, which are then executed using Spark's distributed processing framework.

The streaming concept has been divided into micro-batching processing technique or non-batch processing techniques. Spark Streaming solution provides micro-batching of the unbounded stream. It incorporates stream processing via

[1]http://zeromq.org/

Table 2: Attributes based comparison of Open-source data processing solutions

Attributes	Spark	Storm	S4
Framework	Micro-Batching with Batch Processing using Core Spark	Stream Processing + Micro-Batching using Trident	Actor Programming Model
Implemented in	Scala	Clojure	Java
Application Language	Java, Scala, Python, R	Java, Clojure, Scala, Python, Ruby	Java, Python, C++
Stream Primitive	DStream	Tuples	Events
Stream Source	Network, HDFS	Spouts	Network
Computation or Transformation	Transformation, Window Operations	Bolts	Processing Element
Persistence Entity	foreach RDD	Bolts	Control Messages
Reliable Execution	Exactly once	At least once	–
Fault Tolerance	Tiny bits loss possible, Require HDFS for fully fault tolerant	Tuples replayed, Guaranteed delivery	New Node begin from snapshot
Latency	Few Seconds	Sub-Second	Few Seconds
Developed By	Conceived by AmpLab Berkely, Now Apache incubation project	Conceived by BackType/ Twitter, Now Apache incubation project	Initially conceived by Yahoo!, Now Apache incubation project

short interval of batches and provides a linear streaming solution, which is suitable for existing batch processing infrastructure. Storm and S4 both adopted non-batch processing techniques. Storm also provides micro-batch processing using Trident APIs. Apache S4 is entirely focused on real-time stream processing and does not support micro-batch processing. The attribute based comparison is performed in Table 2 between Storm, Spark, and S4. Table 2 highlights different aspects of these solutions, which are compared in the context of processing model, data pipeline, latency, fault tolerance, and data guarantees.

Apache Spark is a computing framework provided by Berkeley Data Analytics Stack (BDAS). In the next section, we explore BDAS and relative products or software frameworks available in BDAS. We learned more about the internals of Apache Spark and Spark Streaming in the next section.

2.3 Berkeley Data Analytics Stack

BDAS is an open-source stack of software components for data analytics [2]. It is developed by AMPLab, UC Berkeley[2] and widely supported by the open-source community as well as industry. Over the years, BDAS included various third party software components and extended support for integrating with other software frameworks. Figure 4 presents different layers of BDAS components: resource virtualization, storage, processing engine, and interfaces.

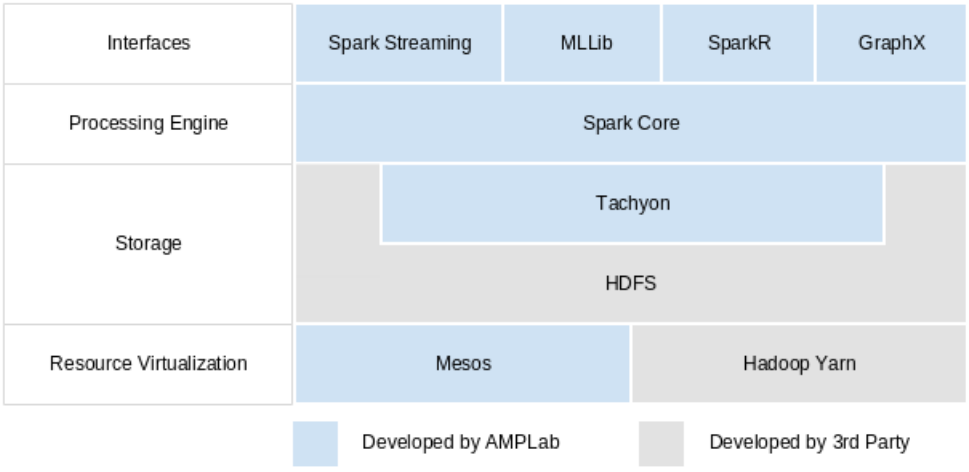


Figure 4: Software components in Berkeley Data Analytics Stack.

[2]https://amplab.cs.berkeley.edu/

At the lower layer, BDAS consists of general-purpose cluster resource manager called Apache Mesos[3]. BDAS also supports integration with third party Hadoop Yarn[4] cluster manager. Cluster resource manager is a platform for management of different nodes in a cluster. It can run on one or more cluster nodes. It provides high-level API for job scheduling and resource allocation in cluster nodes. Storage layer involves Tachyon and Hadoop distributed file system (HDFS) as file systems. Tachyon[5] is a distributed in-memory file system developed by AMPLab. HDFS is the most used third party file system in applications. HDFS is a highly fault-tolerant distributed file system designed to provide high data access throughput for large data sets [3]. Spark core is the processing engine or computing framework used in BDAS. Interfaces include in-house developed software components and third-party modules built on the top of the processing engine. Spark Streaming, SparkR, GraphX, MLlib are the names of several interfaces. Application layer involves applications that run over the BDAS using one or more interfaces. User applications run over BDAS using one or more interfaces.

2.3.1 Spark and Spark Architecture

Spark is an open-source general-purpose cluster computing platform for distributed data processing. Spark provides a unified platform for batch processing, stream processing, and interactive computations. Spark provides API in Java, Scala, R and Python to write an application for Spark. It is also possible to extend existing interfaces provided by Spark to model it as per application requirements. For this thesis, we focus on Scala as a primary language of discussion in future context.

Zaharia et al. [84] presented Resilient Distributed Dataset (RDD) as the core logical unit of Spark. RDD is an immutable partitioned collection of data elements. The inherent partitioning in RDDs assists in parallel computation in Spark. The properties of the RDDs are described as follows:

- Immutability. The RDD is read-only collection of data elements. This allows RDD to store or process them across different worker nodes in Spark.
- Laziness. The RDDs are lazy in nature, such that, they enclose all the information required to build their elements, but accomplish them only when

[3]http://mesos.apache.org/

[4]https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[5]http://tachyon-project.org/

required.

- Lineage. The RDDs remember their evolution by keeping track of operations applied to them. This information helps in easy recovery of lost RDDs in the case of failure. Therefore, the lineage information of RDDs provides fault-tolerance in Spark.

- Data locality. Spark takes into account location of data while distributing computations across worker nodes. It provides user interfaces to manage data locality for computations.

Spark provides fast in-memory computation using RDDs. It includes a set of operation such as transformations or actions that can directly be applied on RDDs. Figure 5 presents actions and transformations on RDDs.

- Transformations. They are applied on RDDs to create new RDDs from existing ones. Transformations are evaluated lazily by Spark and saved in the lineage graph of RDDs. Operations such as map, reduce, and filter are an example of transformations. For example, a map operation on RDD will wait for actions to materialize the execution of the transformation. Table 3 presents commonly used RDD transformations.

- Actions. They are applied on RDDs to materialize execution of lineage graph and return instant values. Actions are not lazy in nature, but they are saved in the lineage graph of RDDs. Actions always trigger materialization of interim transformations applied on RDDs to return particular result values. Operations such as collect, count and save are examples of actions. Table 4 presents commonly used RDD actions.

The inherent programming model of Spark provides much wider applications as compared to other existing models. It is as well more effective than existing programming models and provides fault tolerance. Zaharia et al. [84] proved that Spark outperform existing systems such as Hadoop for computations in a cluster computing.

Spark configurations can be categorized into a standalone mode or cluster mode. The standalone mode allows Spark installation on a local machine. The standalone mode also be a cluster mode on a local machine. The cluster mode allows Spark installation in a cluster environment. The cluster is a loosely coupled group of

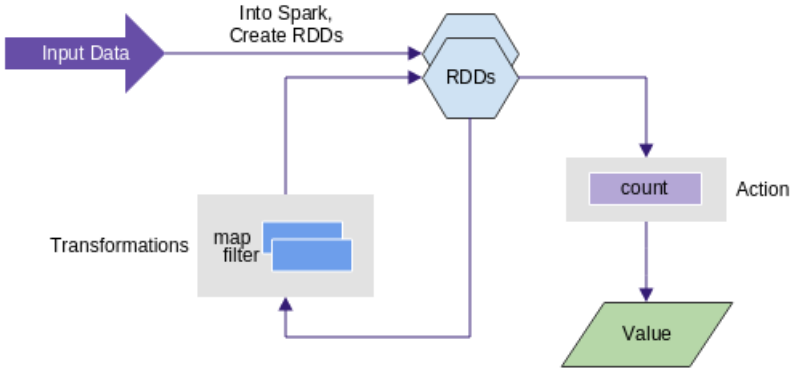


Figure 5: Actions and Transformations on RDD.

machines connected over a local area network. Cluster manager orchestrates a group of machines and ensure that machines work together as a single coherent system. In computer networks, we consider computer machines as nodes in a network. If some machines or nodes in a cluster fail, other nodes in the cluster share the computation task among them. The cluster mode of Spark is shown in the Figure 6.

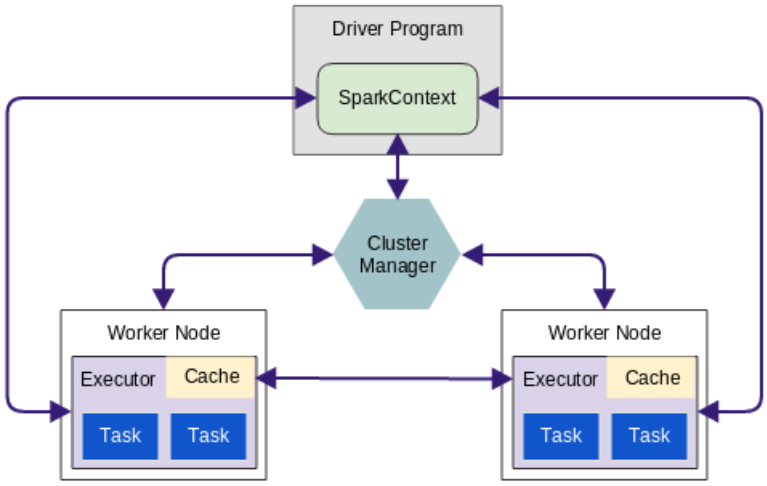


Figure 6: Overview of Spark architecture.

Each application in Spark runs as a collection of processes in a cluster. Spark cluster consists of a master node and more than one worker nodes. SparkContext is the main entry point to the Spark functionality that connects to Spark cluster [4]. SparkContext initialize Spark environment and coordinate the operations in the application. The main program of the application that holds SparkContext is also known as Driver program. Driver program submits processing request to

the master node. Master node is a resource manager, which manages resources for an application. Master node in Spark assigns jobs or computation tasks to the worker nodes. Worker nodes can have more than one executor. The executor is a process, which runs on a worker node. An executor can have more than one physical machine core. Cores of a machine are also defined as slots for a particular executor. Based on the number of slots or cores, an executor can perform some tasks. Task is the smallest unit of work in Spark [51]. Depending on the application, the operations such as transformations or actions defined on a RDD are divided into tasks. Each of these computation tasks is assigned to executors by the master node. Executors perform computations and store results in worker nodes. On completion of computations assigned to executors, the master node collects the results from worker nodes. In this way, Spark architecture enables parallel computation in a distributed environment on a set of cluster nodes.

Table 3: Common RDD transformations available in Spark [6]

Transformation	Defination
map(*func*)	Apply function *func* on each item in a collection to create new dataset
flatMap(*func*)	Similar to map, but apply function *func* to produce flatten sequence of items
filter(*func*)	Apply boolean function *func* on each item in a collection and returns items in true
union(*dataset*)	Returns new collection that contains union of items in source data set and data set passed as argument.
intersection(*dataset*)	Returns new collection that contains intersection of items in source data set and data set passed as argument.

2.3.2 Spark Streaming

The real-time analysis requires system to accept a stream of data in real-time. BDAS includes Spark Streaming as an interface to support real-time processing of the incoming stream of data. Spark Streaming is an extension to Spark API and performs computations using Spark core. In Spark Streaming, the incoming stream

Table 4: Common RDD actions available in Spark [5]

Action	Defination
reduce(*func*)	Apply a function *func* to aggregate items in data set.
collect()	Returns items as an array collection to the driver program.
count()	Returns total number of items in a data set.
take(n)	Returns first n items as an array collection.
first()	Similar to take(1), returns first item of data set.
foreach(*func*)	Apply a function *func* on each item and update items in a data set.
saveAsTextFile(*path*)	Saves items in a data set to a text file (or text files) in a specified filesystem.
saveAsObjectFile(*path*)	Saves items in a data set to a object file using serialization in a specified filesystem.

of data received over a network is grouped together to form small batches. These relatively small batches are called micro-batches. Spark Streaming receive these micro-batches and buffer them to the memory of workers nodes of Spark. This discretized approach to handle streaming data into micro-batches provides a unified experience of batching and streaming within a single platform. It also enables Spark Streaming to utilize the same memory abstractions (RDDs) as Spark core engine [80]. This allows using the same programming model to write the applications using transformations and actions provided by Spark core. This generalization of handling streaming data as micro-batches leads to sub-second latency in Spark Streaming. Similar to batch processing models of Spark, it offers strong consistency, low latency and fault tolerance using parallel recovery protocol. Figure 7 presents an overview of Spark Streaming architecture.

Similar to RDD in Spark, DStream or Distributed Stream is the fundamental abstraction of Spark Streaming [85]. DStream is a collection of RDDs, and each RDD represents a micro-batch of streaming data. Each RDD contains a stream of data acquired in a specific time interval as presented in Figure 8. This programming abstraction allows interoperability between batch processing and

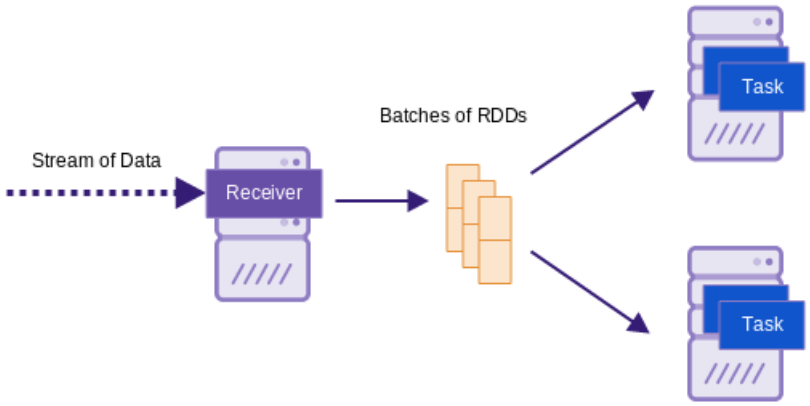


Figure 7: Overview of Spark Streaming architecture.

stream processing in Spark. Spark provides an interface to apply Spark functions on each RDD of DStream. It is also possible to perform interactive queries on micro-batches stored in the memory of worker nodes.

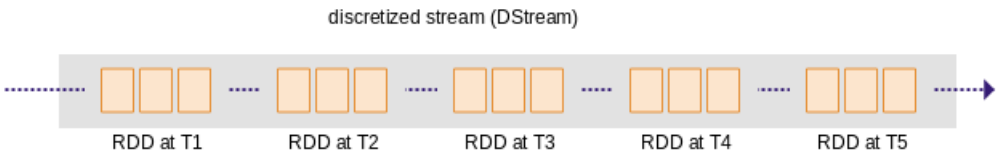


Figure 8: DStream as a collection of RDDs.

In Spark Streaming, the incoming stream of data is received by the receiver in chunks and divided into blocks of data. The time interval at which data is received is known as *block interval*. The default value of block interval is 200 milliseconds [25]. These blocks of data received from a network by the receiver are grouped together into micro-batches. The time interval to form these micro-batches refers to *batch interval*.

A receiver in Spark Streaming provides a custom interface for receiving the incoming stream of data from various data sources. A receiver can be considered as a long running task that receives data streams and executes inside an executor of the worker node. Receiver divides the incoming stream of data into blocks and keeps them in memory. These data blocks are also replicated to other executors to provide fault tolerance and reliability. Based on the batch interval defined in the application, the driver launches tasks to process data blocks obtained by the receiver.

In the case of failure of an executor which is running a receiver, the receiver and all the related data blocks will be lost. In such scenario, the driver tries to restart the receiver on another executor and restart the tasks on the available replicated data blocks. Spark handles this scenario internally, and no configuration is required from the user end. In the case of failure of the driver, all the executors linked to the driver will also fail. In such scenario, all the computations on the received data blocks will also be lost. Therefore, we need to perform a proactive configuration. In this case, we need to enable DStream checkpoint, so that Spark will periodically save the Directed Acyclic Graph (DAG) of DStreams into HDFS. If we have DStream checkpointing, Spark will try to restart driver from checkpointing information saved in HDFS. As soon as the driver is up, it will launch new executors and begin receiving the data stream via receivers. To prevent loss of in-memory data blocks when the driver is restarted, the application requires Write Ahead Logs (WAL). WAL synchronously saves received data blocks to the HDFS. Such recovery measure also requires configuration and code handling to provide fault tolerance and reliability. Figure 9 presents data persistence in Spark Streaming.

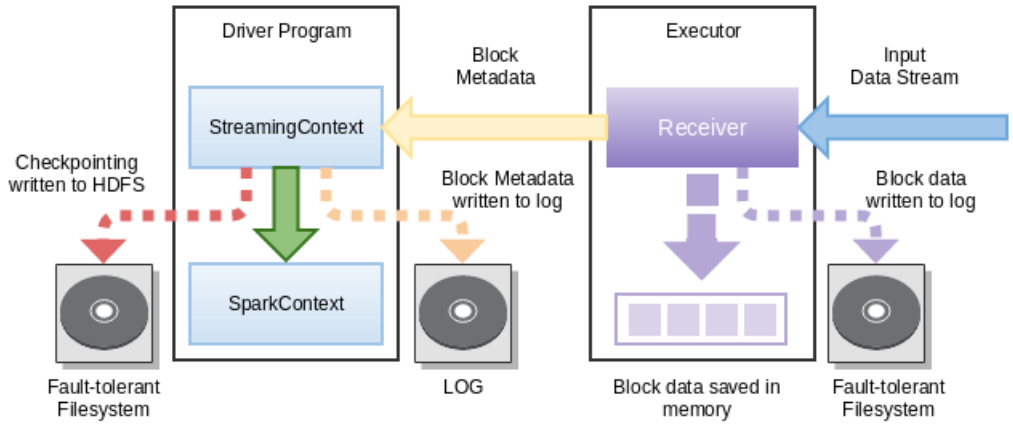


Figure 9: Data persistence in Spark Streaming.

2.4 Data Collection

The growth in healthcare and digital technology provides multiple interfaces to collect physiological signals from an individual. The diversity of these interfaces and lack of standards often lead to interoperability issues between devices. Therefore, bridging the gap between devices is an important aspect to collect valuable medical

data. Lab streaming layer (LSL) [20] together with Kafka [57] provide a solution for bridging the gap between different devices. LSL and Kafka enable data collection from a diverse set of heterogeneous or homogeneous devices, which can be later consume by data processing framework.

Lab Streaming Layer. LSL provides lower layer open-source interface to collect time series signals from various devices and handles time-synchronization. LSL provides interfaces in various different languages such as C, C++, C#, Java, Python and MATLAB [20]. It helps in multi-modal data acquisition such as EEG, EMG, EOG, heart rate, from various hardware providers available over a network.

LSL provides a lower layer abstraction that includes stream outlets and stream inlets.

- Stream Outlets can be considered as consumers, which consume data from different devices, in a sample-by-sample or chunk-by-chunk format. It can consume single or multi-channel data, with regular or irregular sampling data rate.
- Stream Inlets can be considered as receivers, which receive data samples in order and reliably from subscribed outlet. It also provides stream metadata in XML format. Resolver function helps in resolving the streams, based on context-based queries. It involves stream resolution based on name, source identifier or content type.

LSL provides time synchronization for a received data over a local network. It achieves time accuracy of sub-milliseconds for a received data [20]. It also provides an interface to add user-supplied timestamp for a received data. The time synchronization in LSL is based on Network Time Protocol[6] (NTP). LSL provides both timestamp and clock offset to the receiving computer. It requires a receiver to perform mapping of timestamp with clock offset to produce accurate timestamp for the sample.

Apache Kafka. Apache Kafka is an open-source distributed messaging system written in Scala. It provides a unified interface to aggregate different data streams in real-time. Kafka follows a publish-subscribe pattern, in which the publisher sends messages, and the subscriber receives those messages independently of each other. In Kafka, publishers are called Kafka producers and subscribers are called

[6]https://tools.ietf.org/html/rfc5905#section-14

Kafka consumers. Kafka producer is a process that generates messages to the topics, whereas Kafka consumer is a process that subscribes to a particular topic and receives messages from the subscribed consumer group. Kafka organizes the incoming stream of messages and records them into groups called topics. Internally, a topic is organized into multiple partitioned logs on Kafka brokers [57]. Each partitioned log is an ordered collection of messages. Messages are continuously appended in an incremental sequential log order to a commit log. Kafka divides the topics into partitions for scaling and redundancy. Figure 10 presents Kafka topic as a partitioned log.

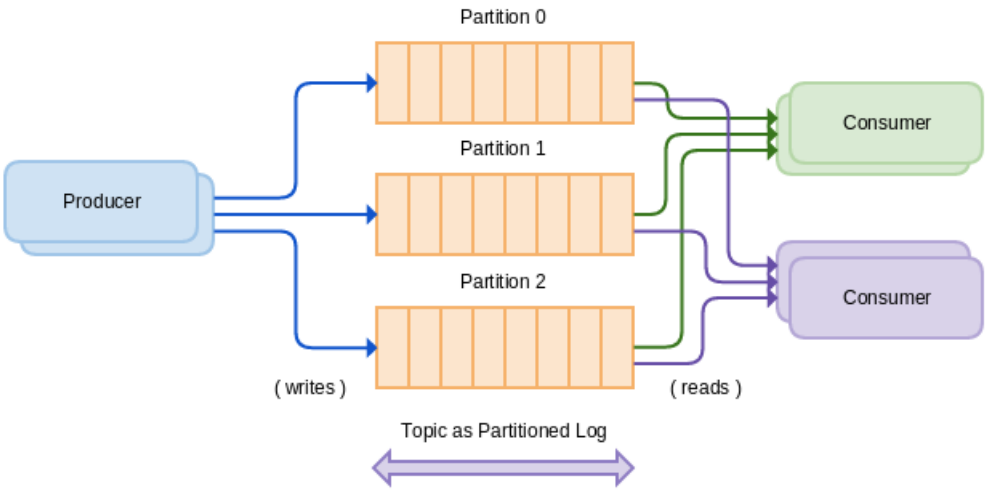


Figure 10: Kafka topic as a partitioned log, writes from producers and reads by consumers.

In a cluster environment, Kafka can have more than one nodes or servers, which is also known as brokers [57]. The design of distributed messaging system provides fault-tolerance and durability by persisting messages across a disk in a cluster environment. In this way, Kafka cluster keeps all the published messages for a particular period of time. These messages might or might not be consumed by the consumers associated with Kafka. Figure 11 represents the high-level abstraction of Apache Kafka.

Kafka provides an unbounded buffer for both streaming and non-streaming data. Internally, Kafka maintains data in structured immutable commit logs or write ahead logs. Write ahead logs or commit logs is the core abstraction of Kafka. Producers or data sources send a stream of data that are appended to these logs.

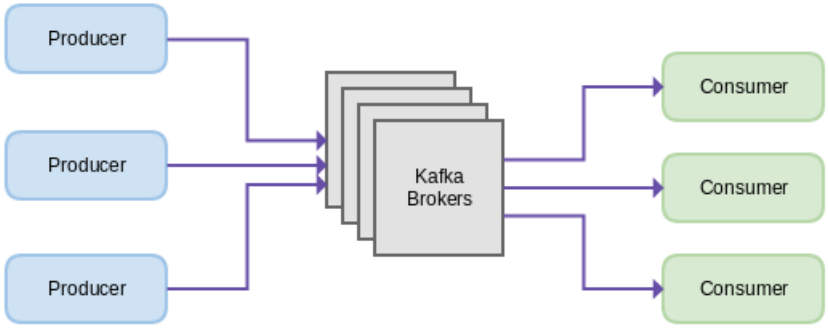


Figure 11: High-level abstraction of Kafka involves producers, centralized cluster of brokers, and consumers.

Multiple consumers from an application in Kafka are grouped together into a logical unit called consumer group. Kafka automatically deduces the order of distribution of messages in the partitioned logs of a topic to instances of consumer groups. The instances of consumer groups can consume or read these updates from partitioned logs. Each consumer can have its own pointer to read data from these logs independent of other consumers. Therefore, multiple consumers can read data in parallel and in ordered way from Kafka.

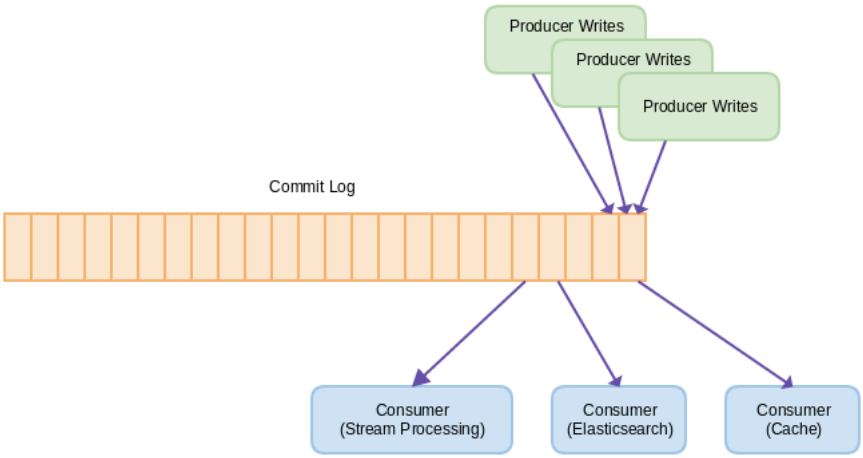


Figure 12: High-level abstraction of Kafka commit log.

The logical organization of partitioned logs and consumer groups in Kafka provides even load distribution of messages to instances of consumer groups. The records or messages in Kafka are ordered using a log sequence number called offset. Kafka

maintains the state of the system using the notion of the log sequence, which is a logical timestamp to the organized records in the commit log. Figure 12 represents the high-level abstraction of commit logs in Kafka.

Kafka provides a dynamic interface to add or remove producers or consumers. It scales from zero to N number of consumers or producers in a distributed environment [24]. A reliable stream of commit log updates, low latency and fault tolerance in a cluster environment make Kafka a reliable interface for data ingestion.

2.5 Data Indexing and Visualization

In various domains, data is constantly generated from different sources and injected into the system. Over the period, the acquired data and processed data by the system, quickly grows into a massive data set. The quick growth of data might lead to a large volume of stale data. Under this scenario, the organizations need tools that can retain the valuable insight of data. The valuable insight is possible, if we can index data in a system and perform good visualization. Therefore, the analytics around data and ability to have valuable insight makes it more beneficial for the domains. Elasticsearch together with Kibana are tools that provide a solution to the growing demands of data handling, analytics and visualization in an organization.

Elasticsearch. Elasticsearch [8] is an open-source distributed search and analytics engine. It was developed by Shay Banon in 2010. It is based on Apache Lucene project, which is an open-source text search engine developed in Java programming language. It provides an interface for smooth integration with existing Spark applications. Elasticsearch provide data indexing along with RESTful APIs that makes it an ideal platform for data analysis. The properties of Elasticsearch are categorized as follows:

- Real-Time Indexing. Elasticsearch allows real-time indexing of saved data. All the data indexed by Elasticsearch is open for near real-time search and analytics. It allows quick insight on the stored indexed data.
- Scalability. Elasticsearch enables horizontal scaling of the cluster with the growth of data. As data grows in a system, more nodes or machines can be added to the cluster as per the demand, and Elasticsearch scales smoothly by taking advantage of new nodes.
- High Availability. Elasticsearch asserts safety and accessibility of data saved in

the system. In the case of failure of nodes in a cluster, it performs re-balancing and reorganizing of cluster nodes to ensure high availability of data.

- Multi-tenancy. Elasticsearch provides multiple independent views of the indexed data. It allows multiple queries such as update or filtering to the different views in a cluster environment. It allows accessibility of stored data via different views as per indexed data configurations.

- Full-Text Search. Elasticsearch provides advanced search interface along with filters to operate on stored data in a distributed environment. It allows easy to use interface to extract stored data via query API.

- JSON Document. Elasticsearch saves the data in structured JSON documents. It allows indexing of all the fields in a dataset. It provides automatic data type inference and schema identification from the dataset.

- RESTful API. RESTful APIs provide a unified experience to query stored data in structured JSON format. The API can be consumed by any application over HTTP and allows easy integration of data from Elasticsearch to web applications.

Kibana. Kibana [13] is an open-source visualization interface and provides seamless integration with Elasticsearch. It is a web-based interface, which provides various types of graphical representations such as charts, graphs, histograms, or maps for visualizing data. The graphical user interface allows interaction with data indexed in Elasticsearch. It provides easy to configure settings to create dynamic dashboards. The dynamic dashboards provide a visual interface for real-time changes in Elasticsearch queries. The visual interface enables discovery of large-scale datasets that helps in locating patterns in data. Apart from flexible visualization interface, Kibana also provides analytics capabilities in the form of mathematical operations that can be applied to datasets. The live indexing and distributed search interface along with good visualization enable real-time decision making of the incoming stream of data using Elasticsearch and Kibana. Figure 13 presents graphical user interface of Kibana with indexed JSON data, and Figure 14 depicts a line graph on two channels.

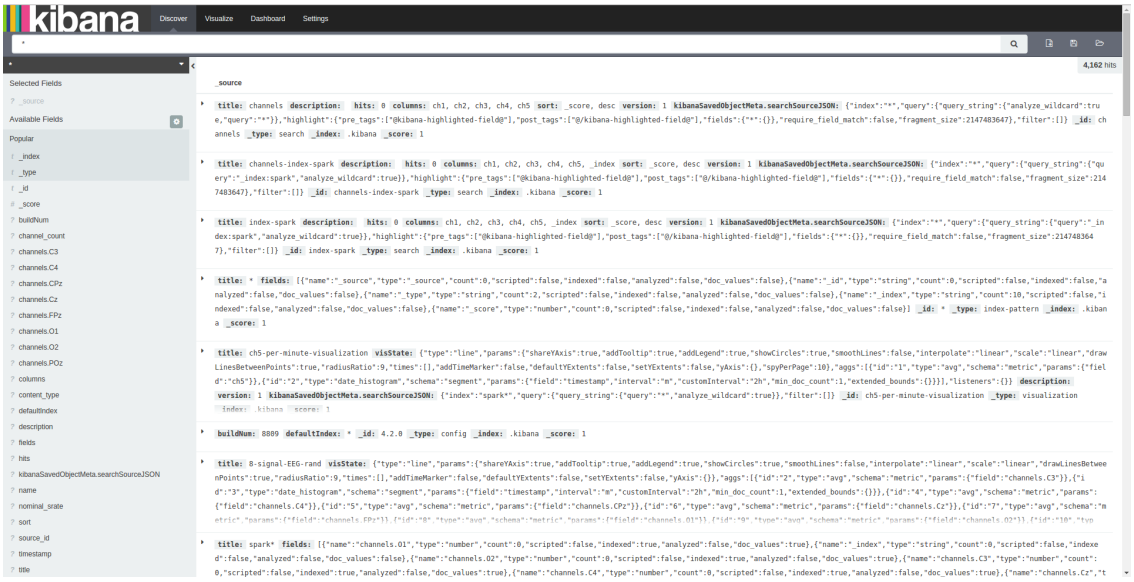


Figure 13: Kibana graphical user interface with indexed JSON data.

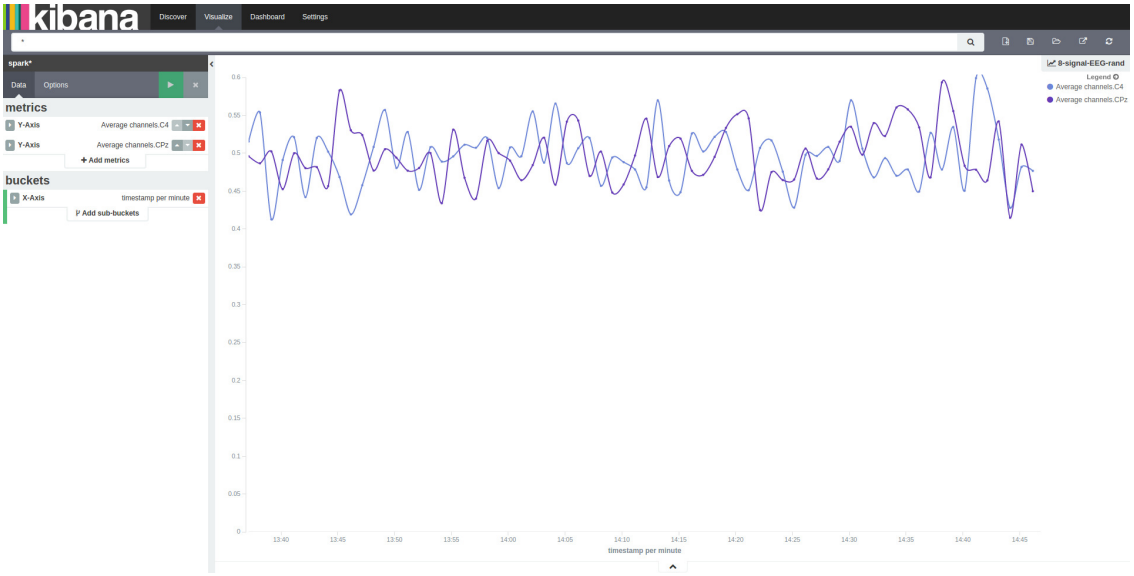


Figure 14: Kibana graphical user interface with line graph on two channels.

3 Research Methodology

This chapter discusses the practical approach employed to carry out the research and software development required to support the experimentation for analysis of physiological signals in distributed environment. The list of tasks involved in the practical approach of our research are defined as follows:

- Understanding of physiological time series signal such as ECG, EEG.
- Requirements analysis for distributed analysis of physiological time series signals.
- Understanding of state-of-the-art solutions for large-scale distributed data processing.
- Preparation of use cases and exploratory data analysis with respect to the use cases.
- Data collection from homogeneous or heterogeneous sources, data pre-processing, data storage, and data indexing.
- Construction of a pipeline to perform distributed analysis of streaming and non-streaming physiological signals.
- Development of programming constructs to process the time series signals in a distributed environment.
- Documentation of steps involved in the research and software development pipeline.

The execution of above tasks requires an organized approach or methodology for the realization of aims of our research. The methodology should support execution of tasks in a sequential, as well as a parallel manner. Tasks, such as understanding of physiological time series signals, requirements analysis, understanding of large-scale distributed systems, preparation of use cases, and development of pipeline, require sequential approach. The order of each task drives the execution of the next task. Whereas, the tasks such as literature review, documentation of study, data collection, can be done in parallel while performing other tasks. The continuous approach also requires constant iterations to improve the research. The methodology should also support the modular approach for the conceptualization

of tasks and rapid prototyping to understand the feasibility of the tasks. Considering the scenario of the research and development aspects involved in our research, we employed a variant of a well-known software development methodology for research and prototyping [58].

3.1 Overview of K|V Methodology

The Kumiega-Van Vliet Trading System Development Methodology (K|V) was introduced in 2006 by Kumiega et. al. [58]. It was developed to provide a systematic business process for research and development of prototype for trading systems. The K|V model combines existing well-known traditional models, i.e., Waterfall model [72], Spiral model [38] and Stage-Gate model [44] into a unique software development paradigm [58].

The Waterfall model is a simple and easy to use staged development model for requirement analysis, system design, implementation, testing, and installation. However, the inherent sequential staged development cycle, rigidity for change requests and weak support for continuous iterations made it less appropriate to apply directly to our research process.

The Spiral model involves a high level of risk analysis. It consists of four phases: planning, risk analysis, engineering, and evaluation [82]. The process model explores alternative solutions and supports rapid prototyping in an iterative manner. It also supports continuous approval and documentation of the modeling process. Considering the number of resources required at the planning phase, there exists a risk for the spiral to keep on growing. Therefore, it is more suitable for a large project with high risk, rather than to small projects such as our research.

The Stage-Gate model for development process involves a set of stages such as discovery, scoping, building the use cases, development, testing, and validation. Each stage or group of stages are separated by a controlled gate. The controlled gate decides the continuous development process after approval from supervisors. The model provides control over an outcome of the stage before moving to the next stage. It ensures quality and continuous development in an incremental order. However, the approval process involved in the process model may cause a delay in the continuity of the project.

The shortcomings described in each of these models are overcome by K|V model by integrating them into a unique research and development paradigm.

3.2 Adaption of K|V Methodology

The K|V model divides the development process into certain stages. The concept of the stage is similar to traditional waterfall model. Apart from stages, the K|V model also includes continuous iteration process to improve the research. The spiral model used in the K|V model consists of four basic steps: research, planning, implementation and test. The continuous spiral cycle is controlled by the gate after four steps. According to the outcome of the stage, the control gate decides if the process requires another iteration cycle, or outcome is appropriate to move to the next stage. Since, K|V model was designed for research and development process involved in trading systems, therefore, we made certain changes to adapt K|V model to our academic research context. The notable changes in the adaptation of K|V model are as follows:

- We use peer review process instead of the hierarchical executive team review process as a stage gate to move from one stage to another.
- After a review process, if we require additional cycle at a particular stage, then all the steps involved in the stage are not mandatory to follow.
- Iteration process is a continuous process to improve the outcome of the research, but the iteration cycle may yield the same result as deliverables of the research.

In our research context, the organization of research into stages, similar to the stages in the traditional waterfall model, enables modularity and provides a direction for finding a solution for distributed analysis of physiological signals in medical telemetry. The spiral model brings iteration process to a stage and improves the quality of deliverables via continuous research and development process. The learning from each stage or each step is applied to other stages or steps to improve the overall quality of the research process. The stage-gate model ensures continual growth with quality deliverables using peer review process and motivates us to be inclined towards the scope of research.

As per K|V model, our scope of research is organized into four stages, and each stage consists of four steps. Each stage may have different steps as per the context of the stage. Each stage produces constructive deliverables that act as an input to the next stage. As per the review process, if we need further improvement in the stage deliverables, we proceed for another iteration cycle. Otherwise, we move to

the next stage. The iteration process may produce minor or major changes to the deliverables of a particular stage. The continuous peer review process streamlined the productivity of timely deliverables from each stage. Figure 15 presents the adaptation of K|V methodology in our research.

3.2.1 Conceptualization Stage

In the first stage, we started our research by exploring existing state of the art systems used for processing of physiological time series signals in medical telemetry. The scope of the research was understanding of physiological time series signals and evaluation of the applicability of distributed approach for their analysis in a distributed environment. The primary focus was given to the problem statement, understanding the factors linked to it, and exploration of methods and tools to solve the problem. The stage involves four spiral steps described as follows:

Domain understanding. The main purpose of this step was to understand various types of physiological time series signals in medical telemetry. We performed intensive literature review to understand the domain. The courses such as Big Data Frameworks [28], Spark Code Camp [29], seminar on Distributed Computing Frameworks for Big Data [31], offered at University of Helsinki laid down the foundation and provided an opportunity to research on distributed computing frameworks. The discussions and informal meetings with Finnish Institute of Occupational Health (FIOH) assisted in understanding of the research scope and existing methods used in the research domain.

Research quantitative methods. Within this step, we studied various quantitative methods that can be used for analysis of physiological time series signals in medical telemetry. Considering the physiological signals, such as ECG, we evaluated the existing methods, such as the average inter-beat interval length, root mean square value by successive differences, for analysis of ECG signals. Apart from that, we also explored the dynamic time wrapping for processing of time series signals.

Distributed processing tools. This step involved exploring the available open source solutions for distributed large-scale processing to make a conceptual model for the analysis of physiological time series signals.

Conceptual model. This step involved studying the feasibility of open source large-scale distributed processing solutions, identified in the previous step, to

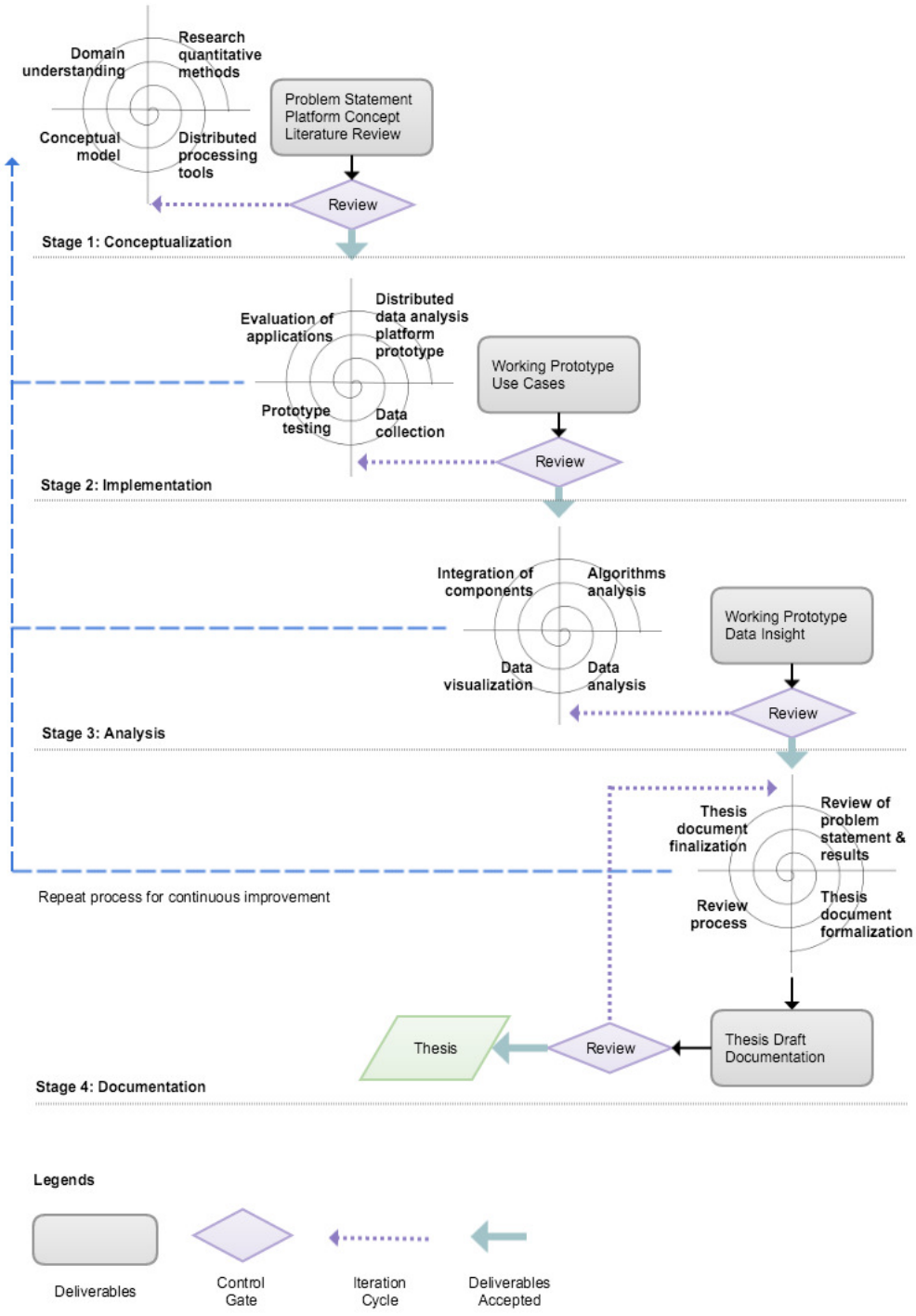


Figure 15: Adaptation of K|V methodology in research and development.

formalize them for the conceptual model.

Deliverables. Formalization of the problem statement and platform concepts are the two important deliverables generated from this stage, which laid the foundation for our research. First two steps contributed towards the problem statement, whereas, the last two steps were involved in the development of platform concepts.

The documentation of literature review is a continuous process in each cycle of each stage and iterations involved in the whole process.

3.2.2 Implementation Stage

The focus of this stage includes requirements analysis, and rapid prototyping of conceptual model learned in the earlier stage. It also includes testing on sampled data and the exploration of a prototype for use cases in the application domain. The main tasks performed during this step are defined as follows:

Evaluation of applications. As per the knowledge obtained in stage 1, the problem statement is revisited to identify the applications that can take advantage of the distributed processing of physiological signals in medical telemetry.

Distributed data analysis platform prototyping. The problem concepts are revisited for practical implementation of the platform prototype. It involves feasibility analysis, installations, configuration, customization and integration of software components to create a coherent pipeline for distributed processing of physiological signals.

Data collection. Considering the privacy concerns related to physiological signals, we collected anonymous privacy-preserving open source data from a reliable online source PhysioNet [49] [30]. PhysioNet is an online web platform, which contains a diverse collection of physiological signals [17]. We also used synthetic time series data for initial evaluation of the prototype.

Prototype testing. This step involves testing the processing of physiological signals from the sampled data with the developed prototype. The testing of workflow process includes various functional aspects such as data collection, data ingestion, data pre-processing, data structuring, data processing, data storage, data indexing, and data visualization.

Deliverables. After several iteration cycles, we delivered a prototype of the

distributed data processing pipeline for physiological signals and exploration of use case applications.

3.2.3 Analysis Stage

This stage involves further evaluation of the prototype for analysis of the data collected in the earlier stage to produce valuable insight based on the use case applications. The main tasks performed during this step are defined as follows:

Integration of components. The modularity and extensibility were one of the primary focus of the developed prototype. Therefore, we tested the modularity and extensibility of the pipeline by integrating available open source software components. We studied various aspects of the involved software components in the pipeline and tried various combinations, such as data ingestion techniques to strengthen the pipeline for future usability.

Algorithm analysis. This step involves the application of various quantitative methods discovered in stage 1 on the collected data. As per the application use case detected at an earlier stage, the library containing algorithms are adapted for processing of physiological signals in a distributed environment. The continuous process of learning is used from various cycles to improve the implementation aspects according to the application use case.

Data analysis. In this step, we use certain algorithms that can be applied to both distributed and non-distributed systems. The quantitative aspects are measured by processing of physiological signals in a distributed and non-distributed systems. The insights obtained from this step accounts for primary results and illustrates the applicability of analysis of physiological signals in a distributed environment.

Data visualization. The insight obtained from the previous step is indexed using Elasticsearch and visualized using Kibana. The indexing helps to keep the processed data valuable and searchable over a period. Whereas, the visualization tool such as Kibana provides flexibility to visualize indexed data into near real-time custom graphs.

Deliverables. At the end, the outcome of this stage is a functional prototype for distributed data processing. We also obtained valuable data insight from this stage, which affirms the applicability of our approach for the analysis of physiological signals in a distributed environment.

3.2.4 Documentation Stage

The documentation of learning was a continuous process across different stages and iteration cycles. The principal focus of this stage was to aggregate the documentation created at each stage and formalize them into a final document. It includes four steps described as follows:

Review of the problem statement and results. We revisited problem statement in this step and iterated over the deliverables obtained from various stages. The central focus was to streamline the research in a single direction and verify the goals targeted at stage 1.

Thesis documentation formalization. We consolidated the documentation created at each stage into a single thesis document. We iterated over the figures, tables, references, structure format, sequencing order and language aspect involved in the thesis document. We organized the information into a single presentable consistent document.

Review process. The thesis document obtained from the previous step was discussed with the supervisors in order to have a valuable feedback to improve the quality of research and document. The review process explored various aspects involved in the documentation and discovered the applicability of thesis context into the future research direction.

Thesis documentation finalization. The reviews obtained from the previous step were included in the thesis to improve the quality of literature. This step involves proofreading and updating the thesis document to finalize it.

Deliverables The final thesis document was the main outcome of this stage.

In our research methodology, we followed various iteration cycles in each stage. We began our research process with a small scope and iteratively moved towards expanding the horizon of our research. The early state prototyping enabled the realization of problem concepts and further refining the problem statement. In this way, the controlled gate approach ensured the quality of each stage, as well as oriented the research in the correct direction. Furthermore, the approach used to accomplish the tasks in a sequential, as well as in a parallel manner, drove the research process to finish in a timely order. The learning from each stage is applied to the next stage, which in overall improves the characteristic of the deliverables. In result, the inherent iterative process involved in the K|V model played a significant role in the overall quality of our research.

4 Distributed Analysis of Physiological Signals

Physiological signals are linked to the physiological changes in the biological systems of an individual. The physiological signals obtained from healthcare systems contribute to a large amount of vital information related to an individual. The signals from different biological systems together form attributes for evaluation of symptoms or diseases. Due to the interconnected nature of biological systems, the aggregated approach is useful to understand symptoms and predict diseases.

On one hand, the advent of technologies allows capturing of physiological signals over a time period. On another hand, the continuous nature of these signals demands context aware real-time analysis. Consequently, the research aspects to analyze physiological signals are addressed using large-scale data processing solution. Under these circumstances, we have developed a general-purpose distributed pipeline for cumulative analysis of physiological signals in medical telemetry. The pipeline is built on top of Spark that performs in-memory computations on a cluster in a distributed environment. The emphasis is given to the creation of a unified pipeline for processing of streaming and stored physiological time series signals. The pipeline provides fault-tolerance guarantees for the processing of signals and scalability to multiple cluster nodes. The pipeline enables real-time indexing of physiological signals and provides visualization of both real-time and archived data. It also provides interfaces to allow physicians or researchers to use distributed and collaborative computing for low-latency and high-throughput signals analysis in medical telemetry.

This chapter introduces the distributed analysis of physiological signals (DAPS) system and the related interfaces. To the best of our knowledge, the DAPS system is the first general purpose open-source distributed platform for cumulative analysis of physiological signals in medical telemetry. Different sections of this chapter provide descriptive information about the DAPS system. Section 4.1 introduces design challenges encountered while conceptualization of the DAPS system. Section 4.2 explains data analysis workflow, which includes data collection and pre-processing methodologies. Section 4.3 explores platform concepts involved in distributed analysis. It includes software components, system architectural and different data ingestion techniques. Section 4.4 provides application programming interfaces of the DAPS system. It includes various application examples. Section 4.5 concludes the chapter with the evaluation of the DAPS system.

In order to understand the conceptual model of the DAPS system, it is important to explore the challenges that derive the design of the DAPS system. We start our discussion with various challenges that influence the design criteria and motivates the development of the DAPS system.

4.1 Design Challenges

Medical telemetry involves various time series signals in the form of physiological signals, which deduce important aspects of a living system. For example, in case of a heart-related problem, ECG is recorded to discover changes in electrical activities of the heart. The analysis of such physiological time series signals requires careful consideration of several factors that can affect the effective outcome of the analysis. Especially the real-time monitoring and analysis of physiological signals comes with various design challenges, which are addressed in the DAPS system.

Heterogeneity. Since different vendors provide hardwares or devices that can assist in measuring these physiological signals, the system which can process these signals should provide an interface to receive these signals. The system requires wrappers that can act as bridges to consume data from homogenous or heterogeneous data sources.

Variability. The physiological signals coming from different sources may vary in time. This aspect leads to synchronization issues. Therefore, a system should support an offset measurement to correctly map time for these signals.

Data Model. Considering the diversity of data sources, the data received by the system should be organized into a structured format. The structured data format enables the easy discovery, singular processing and extensive visualization of the underlying physiological time series signals.

Streaming Aspects. There exists various software components to process real-time signals. The selection of these streaming interfaces to perform real-time analysis of physiological signals requires an evaluation of domain context. An interface which can provide fault tolerant guarantees and exactly one semantic, i.e., which can process data records in an exactly once order, could be an optimal interface.

Dynamicity The system should be coherent, which can provide dynamic addition or removal of data streams coming from different sources. The cumulative analysis of these signals in a distributed environment requires software components that are

inherently distributed in nature.

Extensibility. The system should take benefit from various open-source softwares. The system should consider future need and growth aspects as per context of the domain. The system should be easily extensible to provide a coherent system.

4.2 Data Analysis Workflow

The DAPS system involves various software components, which are written in different programming languages. These software components together form a pipeline to process physiological signals. The synergy between these software components is an important aspect of the pipeline. The different stages of the pipeline such as data collection, pre-processing, data processing and data visualization require a single coherent interface to communicate with them. The interoperability among the different stages of pipelines is an important aspect of the system. The solution requires a common interface or semantics that can be understood by different software components. The interface should support minimalistic structured data types that can be easily serialized or de-serialized during data exchange among the software components.


```
{
  "channel_count": 8,
  "channels": {
    "C3": 0.1688793455371752,
    "C4": 0.5010984930695679,
    "CPz": 0.054803865142659336,
    "Cz": 0.9384443965402033,
    "FPz": 0.9558574770337751,
    "O1": 0.5741773066420076,
    "O2": 0.0023536802159995762,
    "POz": 0.3225907961953479
  },
  "content_type": "EEG",
  "name": "BioSemi",
  "nominal_srate": 100,
  "source_id": "uid1000"
}
```


Figure 16: Synthetic data in JSON format.

Considering the evolution of Web 2.0, JavaScript Object Notation (JSON) appears to be a leading markup language for data exchange. JSON is a de-facto open standard for data exchange between web and mobile applications [60]. JSON is also the most commonly used data transfer standard for RESTful APIs [78]. It is inspired from JavaScript but incorporates limited set of most commonly used

data types [11]. The set of data types enables easy representation of JSON in any programming environments. JSON provides two basic data types: a collection of key-value pair and list of values. Former is considered as an object, whereas later is considered as an array in different programming languages [11].

In the DAPS system, we represent the samples received from physiological signals in JSON data format. The sample represents a single data point, which contains channel information as well as channel data values. As JSON is a de-facto standard to exchange data between different communication media, the JSON format normalizes interaction with other native interfaces and software components used in the DAPS system. Figure 16 presents a synthetic data in JSON format.

During data ingestion in a system, we normalize input raw data stream into structured JSON data format. The structured JSON data format provides singularity to the system. Different software components can parse data format as per their context. For example, Spark provides interfaces for efficient serialization and de-serialization of JSON data format to Scala objects and vice-versa. The JSON data format also enables indexing of data in Elasticsearch, which was further used in Kibana for visualization.

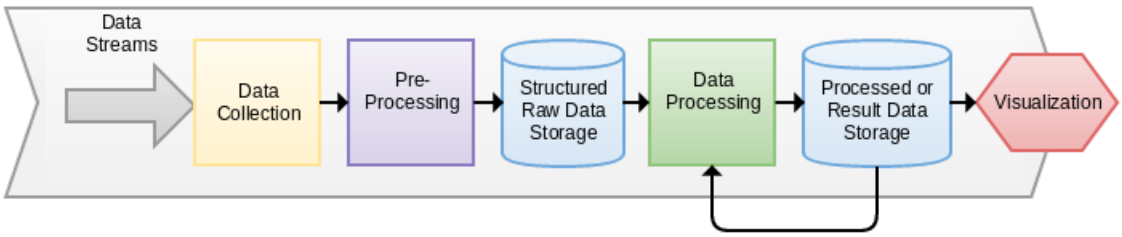


Figure 17: High-level data process flow.

Data analysis workflow involves the collection of data from multiple homogeneous or heterogeneous data streams. It includes homogeneous or heterogeneous healthcare devices and data saved on storage system such as local filesystem or HDFS. Data streams received over a network can be in semi-structured or unstructured format. Therefore, the organization of data in structured format streamline the workflow and enables different components of the DAPS system to process data. Moreover, the stored data can be used again by the system for derivation of different models to map incoming data streams. Figure 17 presents an abstraction of data process

workflow.

4.3 Distributed Analysis Platform Concept

This section introduces the distributed analysis concepts and the DAPS system. The Data analysis workflow introduced in the previous section is aligned with the DAPS system. The proposed system is based on a stack of open-source software components that together form a pipeline for distributed processing of physiological time series signals. There are other proprietary softwares or frameworks that can provide alternate to the proposed model, but the research is inclined towards open-source software components and does not include information about other proprietary softwares.

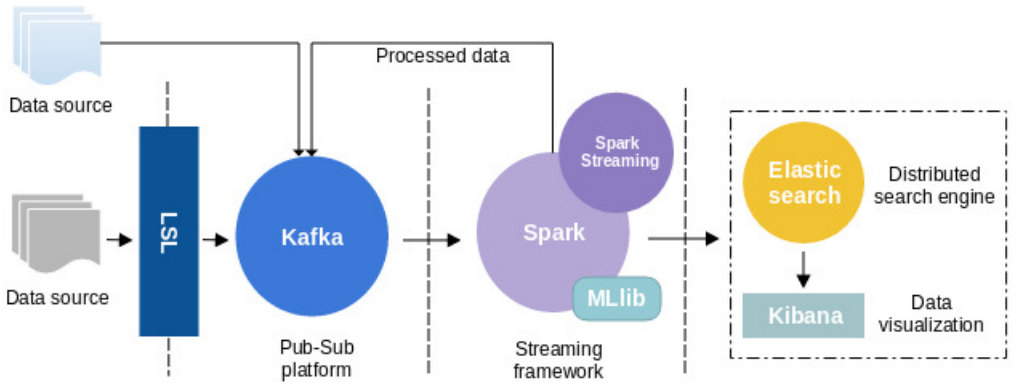


Figure 18: System Architecture of the DAPS system

The challenges identified in the real-time monitoring and analysis of physiological signals are addressed by the DAPS system. The main objective of the DAPS system is to perform cumulative analysis of physiological signals in medical telemetry. The DAPS system involves different layers, and each layer corresponds to process flow defined in section 4.2. Figure 18 presents system architecture of the proposed DAPS system.

Various software components are the building blocks for the DAPS system. The DAPS system is a modular system, which includes Lab Streaming Layer, Apache Kafka, Spark core, Spark Streaming, Elasticsearch, and Kibana. Spark is used as a processing engine, where Kafka acts as an unbounded buffer to handle streams from different resources. Lab Streaming Layer provides lower layer interface for receiving physiological signals. The real-time indexing of data is performed using

Elasticsearch, and Kibana assists in visualization of indexed data. The modularity enhances the extensibility of the system and enables easy integration or disintegration with other modules. Therefore, we can extend system as per context of the domain and requirements.

4.3.1 System Architecture

The focus of the DAPS system is to design a pipeline that can provide a coherent interface for distributed analysis. The components in the pipeline are configured in a way to provide modularity, flexibility and end-to-end scalability for the processing of physiological signals. Components such as Kafka, Spark, and Elasticsearch are inherently distributed in nature, which means the entire system is scalable to multiple cluster nodes. It enables the user to add additional nodes or remove nodes based on usage of the DAPS system. In this way, the DAPS system is a distributed system, which provides a general purpose platform to perform cumulative analysis of physiological signals in a distributed environment. The parallel processing of these physiological signals enables computation load distribution in a distributed environment. For example, measuring the dynamic time wrapping (DTW) distance between channels of physiological signals, involves computing DTW between channels of two time series across different nodes in a cluster environment.

The pipeline of the DAPS system is organized into different stages. Stages represent a logical abstraction of roles involved in the analysis of physiological signals. The software components that are discussed in section 2.3 and section 2.4 laid down a foundation for the stages. The stages can be categorized as data collection, data pre-processing, data processing and data visualization.

Data Collection. The DAPS system can receive signals either from static resources such as storage system, HDFS or from data streams in real-time. There exists a singularity to receive signals from streaming and non-streaming sources and digest them in the same system. Considering the diversity of medical devices, LSL provides a native interface to bridge data received from these sources to Kafka. The devices that are not supported by LSL can also be supported by the DAPS system using a wrapper, which can bridge data sources to Kafka. Kafka provides unbounded buffer, where multiple data sources can publish their data, and multiple consumers can subscribe to the published data. Kafka enables dynamic addition or removal of data streams and provides decoupling of data sources from processing engine.

Therefore, the DAPS system provides an interface to acquire physiological signals data from different data sources in medical telemetry.

Data Pre-processing. Data collection involves acquiring raw data set of physiological signals from homogeneous or heterogeneous sources. Since the DAPS system receives physiological signals from different data sources, the pipeline requires a common data format to communicate between different software components. In general, JSON is most widely accepted lightweight structured data format for Web 2.0 applications. Therefore, these raw data streams are normalized into a structured JSON data format before storing them into Kafka in the pre-processing stage. The structured JSON data format enables different software components to parse the data format as per their context.

Data Processing. Kafka acts as a central repository of an unbounded resilient buffer, and assists in unifying collections of data streams from various resources. The structured data stored in Kafka is consumed by the Kafka receivers in Spark for processing. Spark performs serialization and/or deserialization of JSON data format to Scala objects and vice-versa. If the data is acquired from static resources such as files or HDFS, then the raw data is normalized into structured formats as per the application programming interfaces. The computations involve applying algorithms provided by the underlying extended Spark library. The computations may involve identifying certain features or discovering patterns in physiological signals or applying machine learning algorithms on physiological signals. After computations, the derived data from Spark is available system-wide that can also be looped back to Kafka as a data stream for future processing.

Data Visualization. The DAPS system includes Spark as a processing engine that can perform both batch and stream processing of data. The cumulative analysis of physiological signals on real-time data streams leads to large-scale data in a system. Over a period, the large-scale data collected and processed by the system may result in a huge collection of obsolete data. Elasticsearch together with Kibana provides a solution to the growing demands of data handling, analytics, and visualization. In the DAPS system, Elasticsearch stores the large-scale data into schema-free JSON documents within a distributed storage environment. Elasticsearch adds dynamic behavior to the underlying documents. Over a period, the document schema can evolve as per the change in the input data streams. The inherent distribution of data across several machines provides scalability to the system. The DAPS system integrates Elasticsearch for real-time indexing of large-scale data in a system. The

indexing enables easy discovery of large-scale data sets. The indexing and rich query semantics together provide advanced searching and retrieval of documents. Elasticsearch also provides RESTful APIs to access indexed data in a JSON format in a near real-time environment. The RESTful APIs can be easily integrated with web interfaces to create extended applications to the DAPS system. Kibana along with Elasticsearch provides a live visualization of data indexed by Elasticsearch. The intuitive graphical user interface of Kibana provides easy to configure settings to create dynamic dashboards. The dynamic dashboards provide a comprehensive interface to visualize large-scale datasets in near real-time environment.

Considering the industry wide acceptance of Spark as general purpose computing platform for large-scale distributed data processing, the usage of Spark justifies the pipeline for analyzing of physiological signals in a distributed environment. BDAS also includes various modules, such as Machine learning (MLlib) [27], Graph processing (GraphX) [26], which can be easily used with Spark. Therefore, building a pipeline with Spark as a processing engine, also provides extensibility to the DAPS system and enables us to write any complex computation using other modules as well.

4.3.2 Data Ingestion Techniques

Data ingestion in the DAPS system involves identifying underlying software components. Data ingestion revolves around various ways to perform data injection in Spark. Since, the emphasis is given equally to streaming and non-streaming incoming stream of data. Therefore, we are focussed on strategies to utilize underlying Spark processing engine efficiently. Spark provides various interfaces such as storage system, HDFS or receivers for data injection. Reading data from the storage system or HDFS is a straightforward process in Spark. However, receiver involves reading incoming streams of data over a network. Receivers are used in the context of Spark Streaming for a real-time stream of data. Since, Kafka provides a coherent interface to connect multiple producers and consumers, therefore, receivers can be easily integrated with Kafka for data injection. Spark also provides direct Kafka integration APIs and uses high-level APIs to manage Kafka. The context of the application and domain requires careful consideration to rely on receivers and/or Kafka for the incoming stream of data. In the context of medical telemetry, we explored various strategies to use receivers with Spark Streaming and identify their limitations, which are discussed as follows.

Multiple Spark Streaming Receivers. Initially, we started with a simple setup of embedding LSL stream listener in a Spark Streaming receiver. Such setup requires a list of unique identifiers (UID) or domain contexts such as ECG or EEG to listen to the incoming streams. We organized a list of identifiers or domain contexts by using a configuration file. Spark application reads the configuration file and creates a set of Spark Streaming receivers to receive data streams using LSL. The configuration file was organized in such a way that the number of Spark Streaming receivers should be equal to the list size. Single Spark Streaming receiver can receive more than one stream or multiple UIDs streams. To receive multiple streams, a single line in a configuration file must have more than one identifiers. As we started working with a number of streams, we realized the number of limitations pertained to such configurations. The number of UIDs or different domain context must be declared in advance in a configuration file before running Spark application. In this configuration, we rely on Spark Streaming receivers. Therefore, a number of Spark Streaming receivers must be less than the number of cores in cluster machines. In other words, a system must have *n+1* cores to process the incoming stream of data, where n is the number of Spark Streaming receivers. If we forget to evaluate system configuration, the system can receive the data streams, but it is unable to process them. In addition, the dynamic extension of the system is not possible. In this configuration, we are limited to receive incoming streams of data from different sources or devices supported by LSL. In Spark, a Spark Streaming receiver is a long-running task. Hence, the resources will not be free, even if some incoming streams are stopped completely. Resources will be set free only when we completely stop the execution of Spark application. Considering the limitation of Spark Streaming receivers due to a number of cores, resources and dynamic extension of incoming data streams at run time, we explored other strategies to achieve the objectives.

Spark Streaming Receivers with Kafka. In this configuration, Kafka receiver was integrated with Spark Streaming receiver. Kafka receiver used high-level Kafka consumer API and continuously received data. Spark Streaming receiver or Kafka receiver runs inside an executor or worker node. The received Kafka data is cached in the memory of a worker node and also replicated via WAL. If data is successfully persisted, then Kafka offset is updated by Kafka receiver in Zookeeper [55]. The metadata linked to the received data and its replicated data on log i.e. WAL locations are used to resume the processing in the case of failure scenarios. Figure 19 presents Spark configuration where Kafka is integrated with Spark Streaming receiver.

In this integration, each Kafka consumer group should be embedded in a Spark Streaming receiver of Spark application. Since Kafka receivers should listen to a particular topic, we created a list of topics for Kafka receiver. The receivers will listen and fetch the data streams from the related topics. Under this setup, we consider the topic as a unique identifier or a domain context such as EEG or ECG, where multiple streams linked to a particular domain context will publish data to the related topic. Based on the cluster resources, we can create single Spark Streaming receiver or multiple Spark Streaming receivers that can listen to a single topic or set of topics via Kafka. Such configuration again leads to the same limitations as we encountered in the previous scenario of multiple Spark Streaming receivers. Since we are not using LSL, the dynamic addition of new data streams to the system is possible via Kafka. It requires careful consideration of context identifier, i.e. topics for Kafka listeners to consume Kafka data.

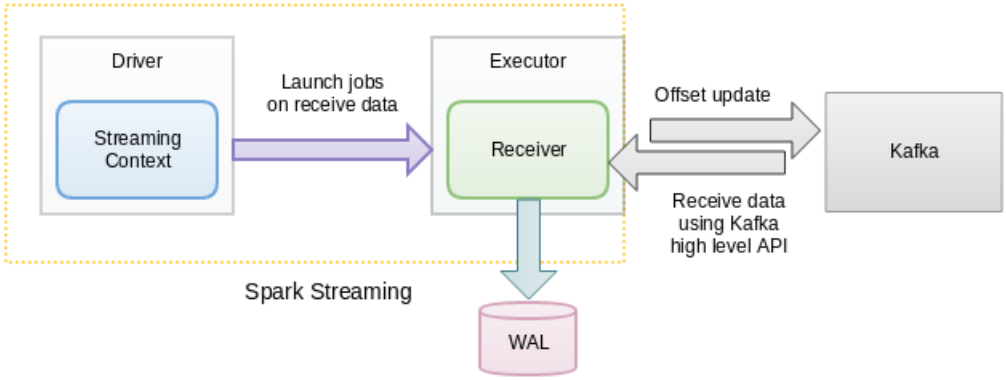


Figure 19: Spark configuration where Kafka is integrated with Spark Streaming receiver [55].

In this configuration, there is a small possibility that may lead to inconsistency in processing records in Spark. Such scenario may arise if the worker node fails after saving the received data to the WAL. As worker node fails, the Kafka receivers will not be able to update the Kafka offsets in Zookeeper. Since the records are reliably persisted to the logs before failure, the system assumes that data is already received up to a particular Kafka offset. On the other hand, since the Kafka offset is not updated by Kafka receiver due to failure, Kafka assume that data is still needed to be delivered of a particular Kafka offset. Hence, after recovery of failed worker node, the Kafka will send the data linked to the same Kafka offset. Due to the limitations of Kafka integration with Spark Streaming receivers and a small possibility of inconsistency, we explored other strategies to achieve the objectives.

Kafka Direct API. In this configuration, the Kafka Direct APIs are used to receive continuous data from Kafka. The Kafka Direct APIs provide consistent view by allowing Spark Streaming to maintain the complete control of offset information linked to a received data [55]. Such control provides consistency in a system and assists in recovery of the node during failure scenarios. It allows the easy replay of data records in a system after recovery. In this approach, offsets are decided before the request for the new batch and these offsets are used to read data from Kafka. The consistency is provided in the system by saving these offsets together with checkpointing. This saved offset information is used to recover from failures. This setup enables Spark Streaming to have exactly-once semantics and provides a fault-tolerance system that can handle stream segments from Kafka [12]. Figure 20 presents Spark configuration with Kafka direct API.

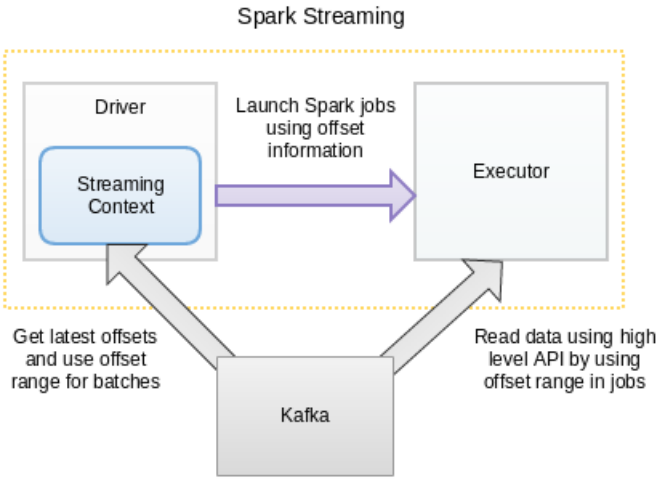


Figure 20: Spark configuration with Kafka Direct API [55].

Since, Kafka receivers should listen to a particular topic, we created a list of topics for which Kafka receivers will listen and fetch data streams. Under this setup, we describe topic as a unique identifier or a domain context such as EEG or ECG, where multiple streams linked to a particular domain context will publish data to the related topic. As we are not using LSL, therefore, the dynamic addition of new data streams to the system is possible via Kafka. It requires careful consideration of context identifier, i.e., topics for Kafka listeners to consume Kafka data. The setup eliminates the use of Spark Streaming receivers or WALs in a system. Therefore, we are not restricted to system cores and long running task resources implications as discussed in previous configurations of using Spark Streaming receivers. Considering the advantages provided by the Kafka Direct APIs, we integrated Kafka Direct API

with Spark Streaming to create a pipeline for processing of a stream of physiological signals in medical telemetry.

4.4 Application Programming Interfaces

The DAPS system includes extended Spark application programming interfaces for analysis of physiological time series signals. The extended interface is written in Scala programming language, but also usable in Java programming language. Scala is a general purpose statically typed and scalable functional programming language, which can run on java virtual machine (JVM). The extended interface includes wrapper around built-in abstraction (RDDs) of Spark. The extended interface provides a descriptive analogy to represent time series signals in Spark. Each physiological time series signal is represented as a collection of samples. A sample is a single data point in a time series data. Each sample includes metadata of time series as well as data points linked to specific channels in a time series.

Developers or researchers can use application programming interface by writing a driver program, which runs on Spark and connects worker nodes in a distributed cluster environment. The driver program creates RDDs as per the data source and performs transformations or actions on RDDs. RDD transformation and actions that are supported by Spark are presented in section 2.3.1. Table 5 presents the APIs developed by us that can be used with real-world time series physiological signals. We further developed ECG and DTW applications based on these APIs that runs on distributed cluster environment.

4.4.1 ECG Application

We present an example of non-streaming data source such as reading data from a file in HDFS. The ECG data is stored on HDFS and is read by Spark into String of RDDs. RDD operations such as map transformation allow mapping of each line in a RDD to the structured data as per the domain context. In the following code example, we are generating a time series RDD from ECG data file on the local file system. In addition, the code executes common ECG operations such as getting all r-peaks for a particular channel, calculating the root mean squares of the successive difference, and average heart rate. The code is presented in the following listing.

```
// RDD[String] : Read data from file
val lines = sc.textFile("data_source_file.txt",8)
```

Table 5: DAPS application programming interfaces

Operation	Meaning
getMovingAverage(channel, windowSize)	Return moving average of a time series for specific *channel* in a window.
getEuclideanDistance(channel, queryRDD)	Return Euclidean distance between two time series of particular *channel*.
getEuclideanDistanceForChannels(channelRDD, queryRDD)	Return Euclidean distance between two time series of multiple channels specified as *channelRDD*.
getDtwDistanceNaive(channelRDD, queryRDD)	Return Dynamic time warping (naive) distance between two time series as per channels specified as *channelRDD*.
getDtwDistanceLC(channelRDD, queryRDD, window)	Return Dynamic time warping (locality constraint) distance between two time series as per channels specified as *channelRDD*.
getDtwDistanceLBKeogh(channelRDD, queryRDD, radius)	Return Dynamic time warping (LB_Keogh) distance between two time series as per channels specified as *channelRDD*.
getMeanHR(ibiSeries)	Return average heart rate for Inter-beat Interval(IBI) series.
getMeanIBI(ibiSeries)	Return average inter-beat interval length for IBI series.
getRMSSD(ibiSeries)	Return root mean square by successive differences for IBI series.
getVariance(ibiSeries)	Return variance for IBI series.
getStdDev(ibiSeries)	Return standard deviation for IBI series.
getAllRPeaks(channel, frequency)	Return inter-beat intervals from the ECG time series.

```
// RDD[SampleUnit] : Organize data into structured format
val timeSeriesRDDs = new TimeSeriesRDD( lines.map { line =>
    val arg = line.split("\t").map(e => e.trim().toDouble);
    new SampleUnit("FileSource", "ECG", "ecg-samples", "local", "10HZ", 1,
        Map("S1" -> arg(2)), Some(new java.util.Date())
    )
   }
 )

// get all R-peaks for ECG channel S1
val rr = timeSeriesRDDs.getAllRPeaks("S1");
// get Average Heart Rate
val hr = timeSeriesRDDs.getMeanHR(rr)
// get RMS Succesive Differences
val rmssd = timeSeriesRDDs.getRMSSD(rr)
```

4.4.2 DTW Application

We present an example of evaluation of the Dynamic Time Wrapping (DTW) algorithm between two time series RDDs. DTW is a distance measurement algorithm, which allows non-linear mapping of one signal to another signal by minimizing the distance between these two signals [68]. DTW iteratively measures the similarity between two time series signals. DTW has been used in many applications such as shape matching, speech recognition or signature recognition.

Single channel DTW on time series signals. In the following example, we measure the similarity between two time series signals. We construct a time series RDD from HDFS data source, whereas we receive other data as streaming signal. The streaming data source is consumed using Kafka high-level APIs provided by Spark. Since streaming signals are received as JSON structured data, we serialize the received streaming data into Scala objects. Figure 21 presents an abstract view of DTW on time series signals received via streaming signal and HDFS in Spark.

```
// read data from file and form RDD[String]
val lines = sc.textFile("data_source_file.txt",4)
// create TimeSeriesRDD of SampleUnit
val timeSeriesRDDs = new TimeSeriesRDD( lines.map { line =>
      new SampleUnit("file","ecg","ecg-sample-1","local","10HZ",1,
          Map("S1"-> line.toDouble), Some(new java.util.Date())
      )
```

```
    }
)

val channel = sc.parallelize(List("S1"));

// handle real-time stream of time series signal
kafkaDStream.foreachRDD {
    rdd => {
        val queryTimeSeriesRDD = new TimeSeriesRDD(  rdd.map{ value =>
            val jsonAst = value._2.parseJson
            jsonAst.convertTo[SampleUnit]
        })
        println("DTW over Streamming time series signal: " +
        timeSeriesRDDs.getDtwDistanceNaive(channel, queryTimeSeriesRDD));
    }
}
```

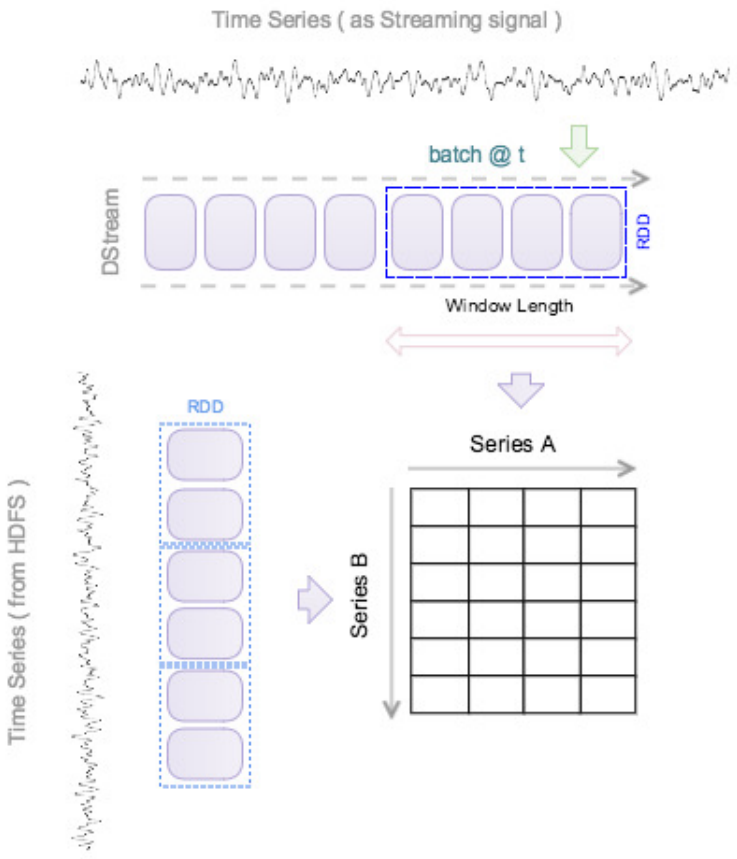


Figure 21: DTW on time series signals received via streaming signal and HDFS.

Multi-channel DTW on time series signals. In the following example, we present the parallel computation of DTW similarity measure between multiple channels of two time series signals. We construct both time series RDD from files on HDFS data source. We use the different variants of DTW such as naive algorithm, locality constraint and lower bound optimization (`LB_Keogh`) [69]. `LB_Keogh` algorithm explored global constraints to optimize windows operation on DTW. `LB_Keogh` in DTW speeds up evaluation of similarity measure between two time series signals [14]. Figure 22 presents an abstract view of DTW on two time series signals received from HDFS in Spark.

```
// RDD[String] // drop headers
val csv = sc.textFile("eeg_motor_mv_source_01.txt").mapPartitions(_.drop(2))
// Organized data into structured format
val timeSeriesRDDs1 = new TimeSeriesRDD( csv.map { line =>
     val values = line.split(",");
     new SampleUnit("PhysioEEGMM","EEG","EEG-Motor-Mv01","Physionet","160HZ",8,
         Map( "FC5" -> values(1).toDouble, "FC3" -> values(2).toDouble,
         "FC1" -> values(3).toDouble, "FCZ" -> values(4).toDouble,
         "FC2" -> values(5).toDouble, "FC4" -> values(6).toDouble,
         "FC6" -> values(7).toDouble, "C5"  -> values(8).toDouble ),
         Some(new java.util.Date())
     )
  }
)
```

Similarly, we read other EEG data source file and organize the data into a stuctured format of RDD of SampleUnits, *timeSeriesRDDs2*.

```
// create RDD of channels to compute DTW
val channels = sc.parallelize(List("FC5", "FC3", "FC1", "FCZ",
                                   "FC2", "FC4", "FC6", "C5"));

// Multi Channel DTW(naive) between time series signals
timeSeriesRDDs1.getDtwDistanceNaive(channels, timeSeriesRDDs2);
// Multi Channel DTW with locality constraint of window size 10
timeSeriesRDDs1.getDtwDistanceLC(channels, timeSeriesRDDs2, 10);
// Multi Channel LB_Keogh DTW between time series signals
timeSeriesRDDs1.getDtwDistanceLBKeogh(channels, timeSeriesRDDs2, 20);
```

The DTW measures the similarity by evaluating the distance matrix between sample points of two different time series. We apply different approaches for building the

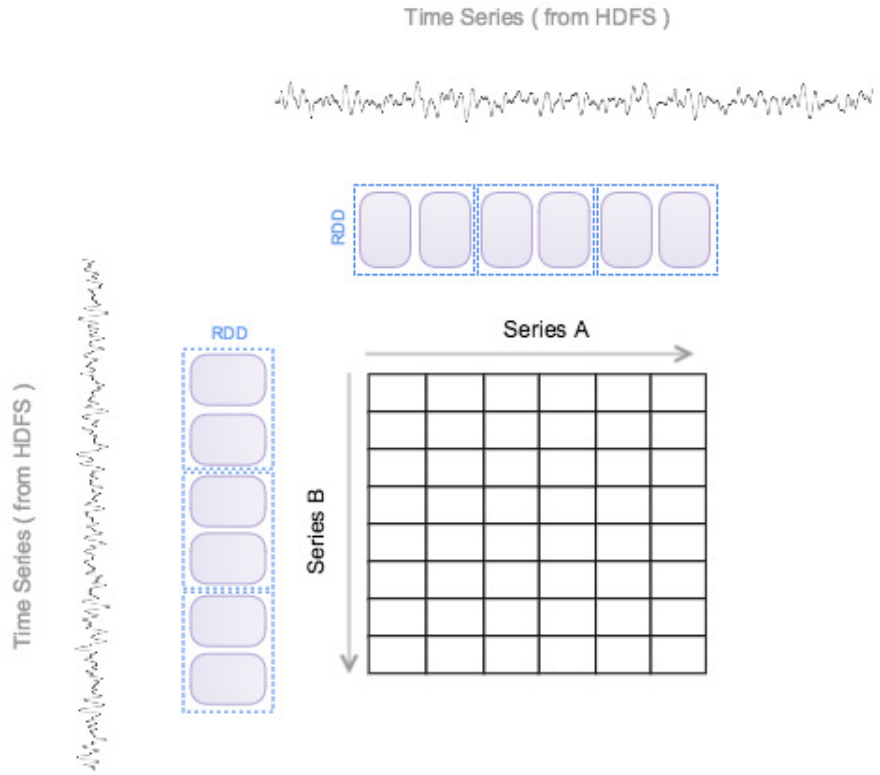


Figure 22: DTW on two time series signals received from HDFS in Spark.

distance matrix depending on the type of data source of the time series signals. The types of the data source can be a file or streaming channels. In case the type of the data source is file, then the time series is constructed using our extended Spark APIs. The data samples collected from two such time series are used to build a distance matrix. DTW can be computed on a single channel or multi-channel time series. For a single channel time series, we create a distance matrix from the data samples of a particular channel of two time series. For a multi-channel time series, we utilize parallel computations on Spark and create a distance matrix between corresponding channels of time series data set. For example, data samples of the *Fcz* channel from both time series can be used to build a distance matrix. Likewise, If one of the time series data set is received by streaming channel, then the samples from that time series are collected in batch intervals. Other time series can be from a file source or stored on a HDFS. We build a distance matrix between all the samples of a stored time series and samples received in a particular batch interval. A distance matrix is always formed between corresponding channels of two different time series.

The DAPS system provides a set of algorithms that can directly be applied to time series signals in medical telemetry. It also includes a comparison between time series signals. We have optimized the provided algorithms for Spark and have written them in Scala. The algorithms include measuring the similarity between time series signals, such as finding dynamic time wrapping between different channels of time series signals using various strategies such as naive method, locality constraint, and lower bound operation. Appendix 1 contains the pseudo code of the different variants of DTW algorithm. The DAPS system also includes algorithms for streaming signals such finding moving average, computing window based dynamic time wrapping. For the ECG signals, we include various operations such as measuring the average heart rate, finding the root mean square for successive differences. Therefore, the extended interfaces for physiological signals together with presented pipeline in the DAPS system enables distributed analysis of time series in medical telemetry.

4.5 Evaluation

We have evaluated the DAPS System for analysis of physiological time series signal in medical telemetry. We have also organized a series of experiments on time series signal for both non-streaming and streaming data sources. We have presented different algorithms for similarity measures using DTW and evaluation of metrics for ECG data sources. The algorithms include single channel as well as multi channel operations on underlying time series signals. As a result, the library developed during this thesis provides flexibility to create time series RDDs from both streaming and non-streaming data sources.

All our experiments are performed on a standalone mode of Spark cluster on a commodity machine. We performed our experiments on University of Helsinki Ukko cluster, as well as on a single machine. The machines or nodes in a Ukko cluster are shared among students of the University of Helsinki. Each node in a cluster consist of 240 Dell PowerEdge M610 machine and has Ubuntu 12.04.5 LTS operating system. Each machine has two Intel Xeon E5540 2.53GHz CPUs with 32GB of RAM. Each node in a Ukko cluster has 16 cores. For our experiments, we used four Ukko cluster nodes. Apart from Ukko cluster, we also used the single local machine for computations. The single machine configuration involves OS X EI Capitan operating system with 2.7 GHz Intel Core i5 processor, 4 cores and 16GB memory. Both worker nodes and master node were established in the mentioned configuration. We used recent versions of stable builds of various

software components involved in the DAPS system: Apache Spark 1.5.2, Spark Streaming 1.5.2, LSL 1.10.2, Kafka 2.10-0.8.2.0, Elasticsearch 2.0.0, Kibana 4.2.0. We used Python 2.7, Scala 2.10 and Java Runtime Environment 7 as an environment to run various components. We also used other open-source libraries for smooth handling of various operations such as Argot 1.0.4 for command line argument parsing, Breeze 0.11.2 for complex computation operations, ScalaTest 3.0.0-M11 for test driven development, and Spray-Json 1.3.2 for JSON parsing. The execution involves command line handling of cluster components such as Spark, Kafka, Elasticsearch and Kibana.

The study was performed to investigate physiological signal analysis in a distributed computing environment. We studied existing large-scale data processing system and validated the applicability of Spark and Spark Streaming for the analysis of physiological signal in medical telemetry. Our contribution involves building a general purpose distributed pipeline and validating pipeline by extending Spark APIs for cumulative analysis of time series in medical telemetry. The pipeline supports analysis of streaming and non-streaming physiological time series signals in medical telemetry. More focus has been given to the real-time analysis of streaming physiological signals in distributed computing environment. We explored various techniques to perform cumulative analysis of these signals in a large environments such as hospitals with many patients.

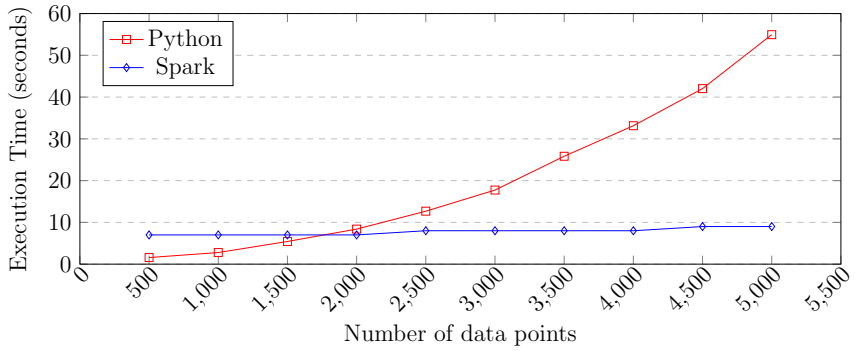
At the specific level, we incline our research work in this thesis on a few time series signals such as ECG and EEG, which provide a starting point for our analysis of physiological signals. The DAPS system includes extended Spark APIs, which provide building blocks for modeling of physiological time series signals in Spark. The algorithms provided in the library can execute computations in both streaming and non-streaming environment. We presented the examples of operations applied to EEG and ECG signals that represent a validation of algorithms provided in the library and process flow in the DAPS system. We also demonstrated the parallel computations between channels of two different time series signals. In addition, we presented the examples of applying operation over an incoming stream of physiological time series signals.

We performed the execution of different variants of DTW algorithm on a number of data points or samples of data sets. Figure 23 presents the execution of different variants of DTW algorithm on a single channel of EEG data set. According to our observations, the naive variant of DTW computation on a standalone mode of

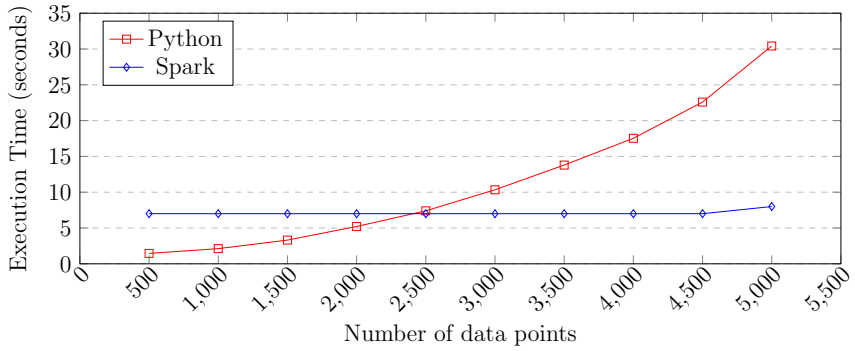
Spark cluster performed slower than Python program for a small data set of fewer than 2000 samples. The processing time is lower in Spark for the computation of more than 2000 data points. Due to the locality constraint, the execution time reduces further in the respective DTW variant. For a data set of more than 2500 samples, the locality constraint variant of DTW computation on a standalone mode of Spark cluster performed faster than Python program. The optimization used by LB_Keogh variant of DTW algorithm on a single channel data set produces approximately constant processing time for both Spark and Python programs.

The execution of different variants of DTW algorithm on a multi-channel data set is shown in Figure 24. We utilized the multiprocessor cores for parallel computations in Python program. In addition, we performed the execution of algorithms on four different channels of EEG data set. We observed that the naive and locality constraint variants of DTW computation on a standalone mode of Spark cluster performed faster than Python environment. However, for a data set with more than 5000 data points, the naive and locality constraint variants of DTW computation leads to memory issues in the computing environment. In contrast, the optimization used by LB_Keogh variant of DTW algorithm produces nearly constant computation time for both Spark and Python programs. Due to the distribution of computations on worker nodes and collection of data from worker nodes in Spark, the processing time increases for a small number of samples in a data sets. As a consequence, the execution of the program on Spark includes a minimal processing time of approximately 6 to 7 seconds. This is apparent from the results presented in the Figure 24.

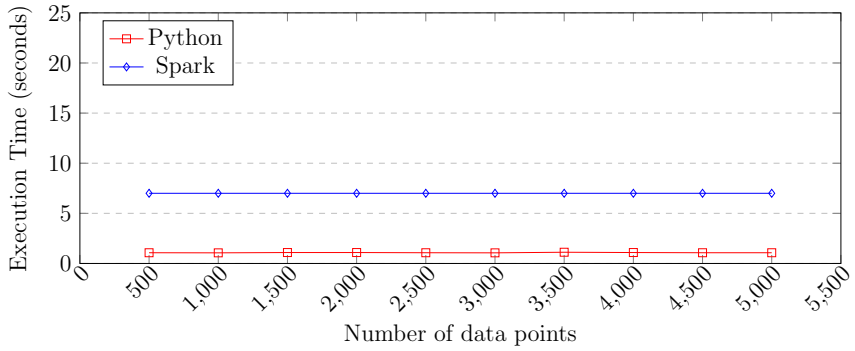
When we operate on large time series signals, the distributed computing cluster handles large data sets due to multiple nodes in a cluster environment. Figure 25 presents the execution of DTW (LB_Keogh) algorithm on a large data set (64-channel EEG signals) of 250000 data samples in a cluster environment. According to the results, the single channel and multiple channel computations on Spark performs faster than Python program. The significance of distributed computation is more apparent when we perform complex operations or operate in parallel on multiple channels of time series signals. In addition, the benefits of distributed computing come when we operate on multiple continuous streams of time series signals. Figure 26 and 27 present DTW (naive) computation on Spark using Kafka direct streaming API's with a batch size of one second in a cluster environment. For Spark streaming, we used the sliding interval same as window length. One of the time series was received from HDFS, while the other time series was received as Kafka stream.



(a)



(b)



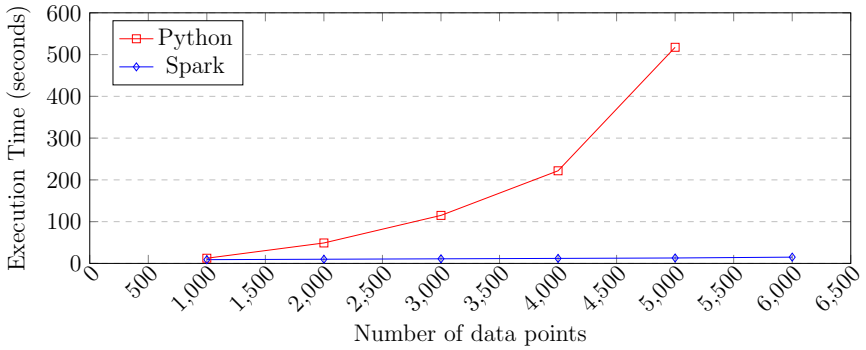
(c)

Figure 23: The execution of different variants of DTW algorithm on a single channel of 64-channel EEG data set. The comparison has been drawn by computing DTW using Spark program and Python program. The computation on Spark involves time series received from HDFS. (a) DTW (naive) execution. (b) DTW (locality constraint) execution. (b) DTW (LB_Keogh) execution.

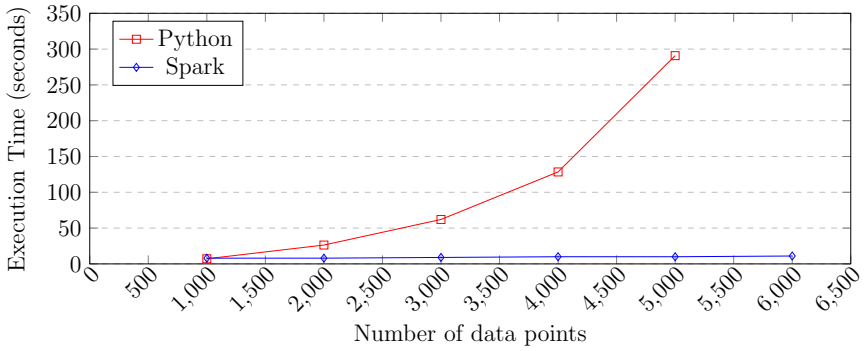
The time series data samples were streamed at a sampling rate of 125. Figure 26 presents the streaming statistics of a single channel time series EEG signals. The computation took an average processing time of 496 milliseconds for processing of last 1000 batches. The statistic presents running of batches of 1 second for 37 minutes 9 seconds. It involved processing of 2224 batches and processed 256789 records. Figure 27 presents the streaming statistics of multi-channel time series (four channels) EEG signals. The computation took an average processing time of 725 milliseconds for processing of last 1000 batches. The statistic presents running of batches of 1 second for 37 minutes 14 seconds. It involved processing of 2228 batches and processed 258180 records. The parallel processing of streaming signals in a real-time environment assists in real-time decision making on time series signals.

The application development on Spark requires the knowledge of programming model provided by Spark. Spark provides a restricted programming model, which consists of RDD and Dstreams. Most of the constructs provided by Spark follow the principle of immutability. Spark does not support nested transformations or actions on RDDs. Therefore, the development of application in Spark requires special attention. Spark provides various tuning parameters to enhance the performance of computations, such as the memory of slave nodes, the number of partitions, etc.

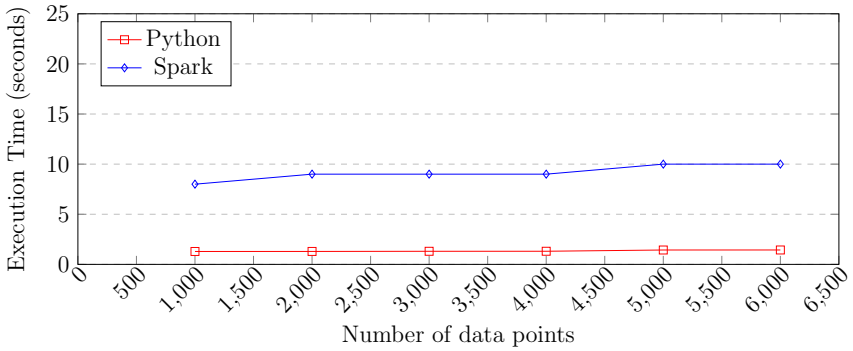
Underlying components in the DAPS system provides low latency, fault-tolerance and exactly-once semantics for processing input data streams. Therefore, computational overhead to process physiological time series signals in the DAPS system is relatively small. The modularity and extensibility of the DAPS system enable integration of presented pipeline with other available software components such as Machine Learning (MLlib), Graph API (GraphX). At the moment, the applications can be written in Scala and Java. The provided library can be easily extended to provide support for different programming languages.



(a)

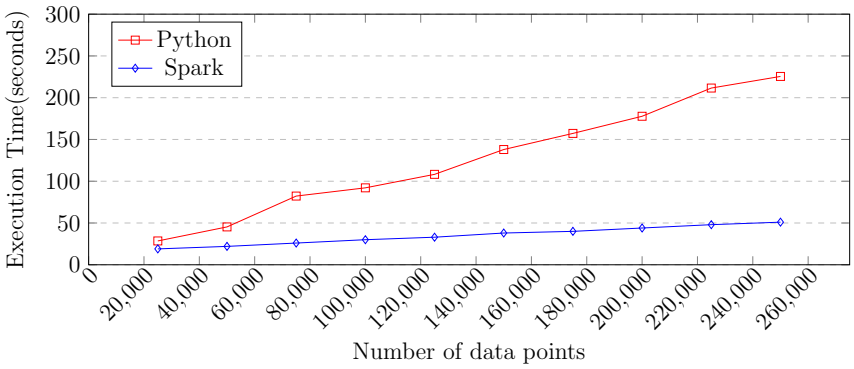


(b)

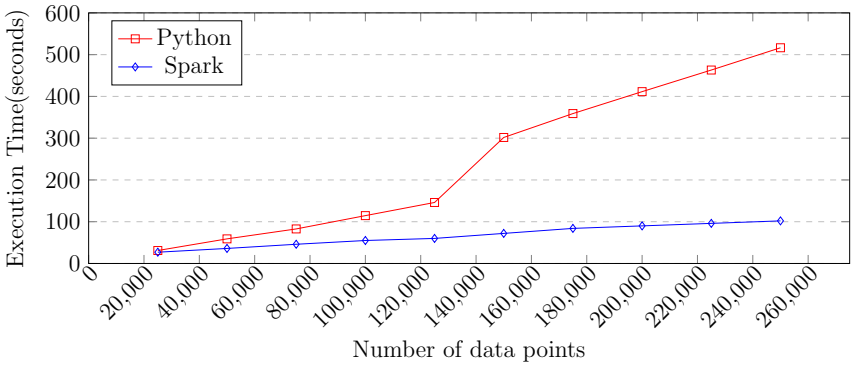


(c)

Figure 24: The execution of different variants of DTW algorithm on a 64-channel EEG data set (sampled at 160 samples per second). The comparison has been drawn by computing DTW on multiple channels using Spark program and Python program. The computation on Spark involves time series received from HDFS. (a) DTW (naive) execution. (b) DTW (locality constraint) execution. (c) DTW (LB_Keogh) execution.



(a)



(b)

Figure 25: The execution of DTW (LB_Keogh) algorithm on a large data set (64-channel EEG signals) in a cluster environment. The comparison has been drawn by computing DTW using Spark program and Python program. Python execution utilizes the cores of the machine and multiple processes execute the computation on channels in parallel. The computation on Spark involves time series received from HDFS, one master node and four worker nodes in Ukko cluster. (a) The computation on single channel of EEG data set. (b) The parallel computation on four different channels of EEG data set.

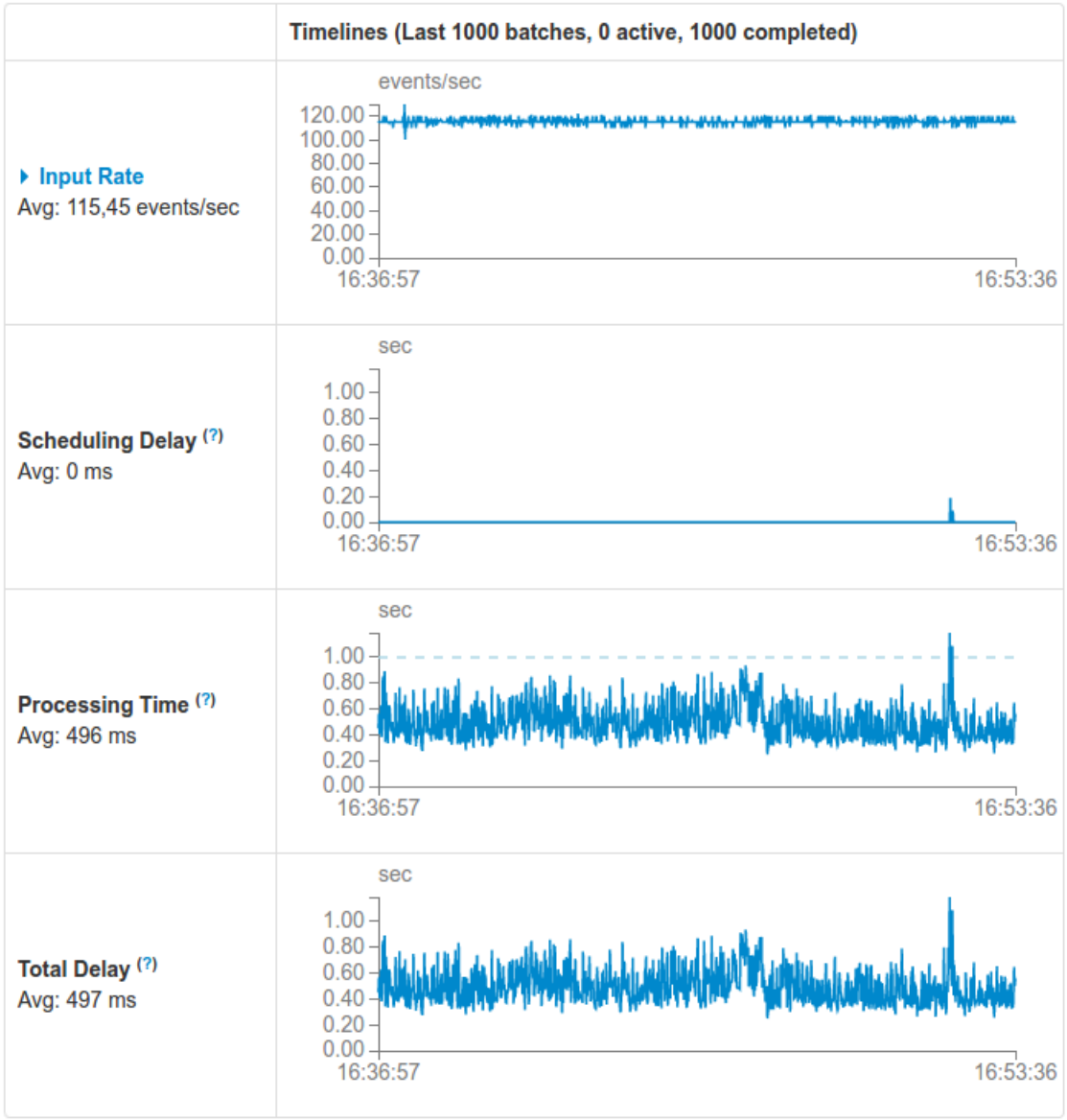


Figure 26: DTW (naive) computation on Spark using Kafka direct streaming API's in a cluster environment. One of the time series was received from HDFS, while the other time series was received as Kafka stream. The single channel time series data samples were streamed at a sampling rate of 125. The streaming statistic presents timeline for the last 1000 batches. In total, the computation time was 37 minutes 9 seconds for batches of 1 second since 16:16:27 (2224 completed batches, 256789 records).

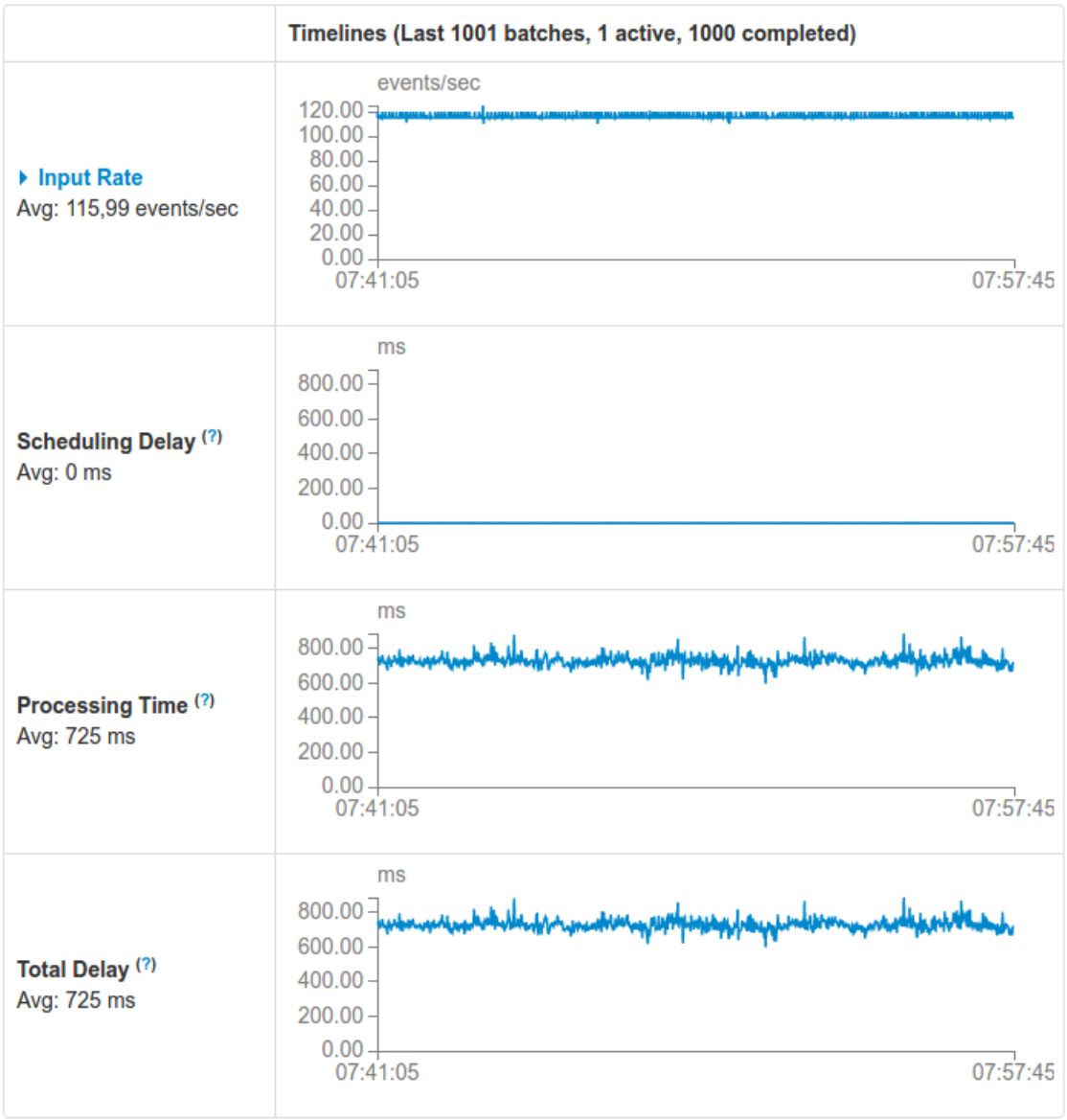


Figure 27: DTW (naive) computation on Spark using Kafka direct streaming API's in a cluster environment. One of the time series was received from HDFS, while the other time series was received as Kafka stream. The multi-channel time series (four channels) data samples were streamed at a sampling rate of 125. The streaming statistic presents timeline for the last 1000 batches. In total, the computation time was 37 minutes 14 seconds for batches of 1 second since 7:20:30 (2228 completed batches, 258180 records).

5 Discussion

The research organized in this thesis enables us to answer various aspects involved in this thesis. We examined existing systems meant for analysis of physiological signals in medical telemetry and discussed whether it is possible to perform distributed analysis of physiological signal in medical telemetry. Chapter 2 provided background information on medical telemetry and explored medical telemetry context related to the research topic. We discussed existing research works and their orientation for the distributed analysis of physiological time series signals.

We investigated existing large-scale data processing systems and their applicability in the research context to provide low-latency, high-throughput and fault-tolerance guarantees. Section 2.2 and Section 2.3 explored distributed data processing using Spark. We studied the utilization of streaming paradigm for the analysis of physiological time series signal in medical telemetry. Section 2.3.3 explored Spark Streaming for real-time processing of data streams. Chapter 3 discussed the methodology applied in our research. Chapter 4 presented a system that can perform distributed computations on streaming and non-streaming physiological time series signals. Section 4.3.1 presented implementation details linked to the pipeline and section 4.2.2 discussed different strategies for data ingestion in a system.

We provided an interface for modeling of physiological time series signals and algorithms to perform cumulative analysis of physiological signals in medical telemetry. Section 4.4 introduced application programming interfaces provided for modeling of time series physiological signals and presented use cases of an extended library for distributed computation on the functional pipeline. Therefore, research aspects linked to the thesis illustrates the breadth of research domain for cumulative analysis of time series physiological signals in a distributed computing environment.

The evolution of computing paradigms provides distributed data processing techniques, which lead to the growth of large-scale data processing systems. The research in this thesis is an attempt to provide convergence of distributed data processing in medical telemetry. The vital information carried by physiological signals and their importance for the healthy functioning of living system justifies the importance of the domain. Although, significant research has been performed in the past to explore physiological signals, less research has been performed to

take advantage of distributed data processing for the analysis of physiological signals in medical telemetry. We contributed in the cumulative analysis of physiological signals using large-scale data processing system. We created a general purpose distributed pipeline and provided an interface to this pipeline by extending Spark APIs. Since, physiological signals contains vital information linked to a biological system, the real-time analysis of these time series signals provides an operational insight about functioning of organs in a living system. More focus has been given to the real-time analysis of streaming physiological signals in a distributed computing environment. The presented pipeline is built to provide coherence and solidarity in the processing of streaming and non-streaming signals. We presented a unified system to perform timely processing of data streams that can be used in large environments such as hospitals with many patients. We have shown promising results by validating the applicability of large-scale data processing system for distributed analysis of physiological signals.

The software components used in the presented pipeline contains Spark as core processing engine for in-memory computations. The distributed computing engine provides various tuning parameters to enhance the system performance. As per the computation context, we have to configure tuning parameters such as resource tuning, parallelism tuning to utilize Spark cluster efficiently.

Tuning parameters such as partition size determines the parallelism in Spark. Partition size plays a significant role in turning of Spark jobs. Spark jobs are divided into tasks, which are grouped together into stages. The partition size of an RDD determines the number of tasks grouped together on a stage and controls the overall processing time. A smaller number of tasks underutilizes the available cores provided by the worker node. Therefore, the stages will be inefficient to take advantage of worker nodes. A large number of tasks leads to more tasks to schedule and execute on worker nodes. Therefore, it will increase the overall processing of tasks. Thus, based on the underlying cluster resources and hardware, the partition size should be configured appropriately to reduce the overall processing and scheduling time.

Spark provides in-memory computations, which produce impressive performance improvements for the computations. However, the processing in Spark is limited by the memory available for the Spark cluster. As per the computation context, we need to configure appropriately the memory tuning parameters to take advantage of the in-memory computations. The identification of memory usage of dataset eliminates

such limitation. The memory consumption can be optimized by building RDD and caching it in the memory. The web interface of Spark cluster provides a monitoring tool to observe the memory consumption of RDD. Therefore, the memory of the underlying cluster resources can be configured as per dataset.

Apart from underlying computation framework and configurations, the development of the DAPS system is an ongoing process. Therefore, the API and analysis algorithms are limited in scope and context. The data visualization tools used in the DAPS system need more attention and require an interface which can be accessible to multiple individuals such as researchers, patients and physicians. As physiological signals are private data to individuals, therefore, we need to address privacy concerns to make the DAPS system reliable for real-time environments.

We presented the DAPS system as a general purpose distributed pipeline for an end-to-end processing of time series physiological signals. The methodologies used in this thesis scratched the surface of analysis of physiological signals in distributed computing environment. The exploratory research utilized in this thesis requires deeper investigations for the analysis of physiological signals in medical telemetry.

Our implementation provides a groundwork for analysis of physiological signals using distributed computing framework, but we would like to extend our work by including research on other physiological signals. In the later releases, we would like to add more features to the library that can provide rich set of constructive operations for physiological signals analysis. We would also like to improve the existing command line interface by providing a web interface to submits jobs to Spark.

The current implementation of the DAPS system supports Scala and Java programming languages. Since, Python is also one of the widely used programming languages in research as well other domains. We would like to extend our DAPS API by providing support for Python programming language. In this way, a wider community could take advantage of the DAPS system.

We would like to keep track of open-source solutions, which can assist in the analysis of physiological signals and we intended to integrate those software components to enhance the presented pipeline. Our aim is to create a distributed, modular, and scalable stack of software components for computations and analysis of physiological signals. We would also like to integrate the presented system in real-time environments such as hospitals, which can provide a quantitative assessment to the DAPS system.

6 Conclusion

We have discussed the distributed analysis of physiological signal in medical telemetry. We reviewed the state-of-the-art in the background information on medical telemetry, and various distributed data processing techniques. We described various systems related to medical telemetry and methodology used in existing interfaces. In particular, the previous research addresses the use of a non-generic non-distributed system for the analysis of physiological signal. Most of the research and tools are focused towards BCI or MATLAB based tools for physiological signals. We also explored software stack provided by Berkeley and learned about related data analysis and visualization tools. Later, we formalized the problem for distributed analysis of physiological signal in medical telemetry. We provided a solution to the problem by proposing a general purpose distributed pipeline. We described software components linked to the proposed system and explained their relation and roles in the suggested pipeline. We validated the pipeline by extending Spark APIs for cumulative analysis of time series signals in medical telemetry.

The implementation of the DAPS system explored various techniques to perform cumulative analysis of physiological signals. Our design is targeted towards cumulative analysis in large environments such as hospitals with many patients. We started our analysis with ECG and EEG physiological signals and used our extended Spark APIs for modeling of these signals. We have shown the applications of distributed computing in medical telemetry. Especially, real-time analysis of streaming physiological signals in a distributed computing environment can assist in proactive monitoring and diagnosis of the functional aspect of organs in a living system. The presented research explored physiological time series signals in medical telemetry and provided new avenues to employ distributed computing in medical telemetry.

Bibliography

1 *Signals And Systems 3E.* McGraw-Hill Education (India) Pvt Limited, 2010. ISBN 9780070672857. URL https://books.google.fi/books?id=XtuS7zCEA10C.

2 Berkeley Data Analytics Stack, Aug 2015. URL https://amplab.cs.berkeley.edu/software/.

3 HDFS Architecture Guide, Aug 2015. URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

4 SparkContext API, Aug 2015. URL http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext.

5 Actions: Spark Programming Guide, Aug 2015. URL http://spark.apache.org/docs/latest/programming-guide.html#actions.

6 Transformations: Spark Programming Guide, Aug 2015. URL http://spark.apache.org/docs/latest/programming-guide.html#transformations.

7 Storm: Distributed and fault-tolerant realtime computation, Oct 2015. URL http://storm.apache.org/.

8 Elasticsearch: Search & Analyze Data in Real Time, Oct 2015. URL https://www.elastic.co/products/elasticsearch.

9 Emotiv EPOC, Oct 2015. URL https://emotiv.com/epoc.php.

10 MIDAS: Modular Integrated Distributed Analysis System, Oct 2015. URL https://github.com/bwrc/midas/wiki.

11 JSON, Nov 2015. URL http://json.org/.

12 Spark Streaming + Kafka Integration Guide, Aug 2015. URL http://spark.apache.org/docs/latest/streaming-kafka-integration.html.

13 Kibana: Explore & Visualize Your Data, Oct 2015. URL https://www.elastic.co/products/kibana.

14 LB_Keogh Homepage, Nov 2015. URL http://www.cs.ucr.edu/~eamonn/LB_Keogh.htm.

15 Mitsar Portable EEG System, Oct 2015. URL http://www.mitsar-medical.com/eeg-system/portable-eeg/.

16 Arrhythmia, Dec 2015. URL https://www.nlm.nih.gov/medlineplus/arrhythmia.html.

17 PhysioNet, Oct 2015. URL https://www.physionet.org/.

18 BCILAB: Open Source Matlab Toolbox for Brain-Computer Interface research, Oct 2015. URL http://sccn.ucsd.edu/wiki/BCILAB.

19 EEGLAB: Open Source Matlab Toolbox for Electrophysiological Research, Aug 2015. URL http://sccn.ucsd.edu/eeglab/.

20 Lab Streaming Layer, Aug 2015. URL https://github.com/sccn/labstreaminglayer/wiki.

21 Apache Spark Streaming Programming Guide, Aug 2015. URL http://spark.incubator.apache.org/docs/latest/streaming-programming-guide.html.

22 DStream: Spark API Documentation, Aug 2015. URL https://spark.apache.org/docs/0.9.2/api/streaming/index.html#org.apache.spark.streaming.dstream.DStream.

23 RDD: Spark API Documentation, Aug 2015. URL http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD.

24 Apache Kafka, Aug 2015. URL http://kafka.apache.org/documentation.html.

25 Configuration: Spark 1.6.0, Jan 2016. URL http://spark.apache.org/docs/latest/configuration.html.

26 GraphX: Spark 1.6.0, Jan 2016. URL http://spark.apache.org/docs/latest/graphx-programming-guide.html.

27 Machine Learning Library (MLlib): Spark 1.6.0, Jan 2016. URL http://spark.apache.org/docs/latest/mllib-guide.html.

28 Big Data Frameworks, Jan 2016. URL http://www.cs.helsinki.fi/en/courses/582740/2015/K/K/1.

29 Spark Code Camp, Jan 2016. URL http://www.cs.helsinki.fi/en/courses/582738/2014/V/K/1.

30 EEG Motor Movement/Imagery Dataset, May 2016. URL https://physionet.org/physiobank/database/eegmmidb/.

31 Seminar: Distributed Computing Frameworks for Big Data, Jan 2016. URL http://www.cs.helsinki.fi/en/courses/58314306/2014/S/S/1.

32 D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003. ISSN 1066-8888. doi: 10.1007/s00778-003-0095-z. URL http://dx.doi.org/10.1007/s00778-003-0095-z.

33 N. Ahmad, D. Hoang, and M. Phung. Robust preprocessing for health care monitoring framework. In *e-Health Networking, Applications and Services, 2009. Healthcom 2009. 11th International Conference on*, pages 169–174, Dec 2009. doi: 10.1109/HEALTH.2009.5406196.

34 D. Apiletti, E. Baralis, G. Bruno, and T. Cerquitelli. Real-time analysis of physiological data to support medical applications. *Trans. Info. Tech. Biomed.*, 13(3):313–321, May 2009. ISSN 1089-7771. doi: 10.1109/TITB.2008.2010702. URL http://dx.doi.org/10.1109/TITB.2008.2010702.

35 P. Aspinall, P. Mavros, R. Coyne, and J. Roe. The urban brain: analysing outdoor physical activity with mobile eeg. *British journal of sports medicine*, pages bjsports–2012, 2013.

36 S. Asur and B. A. Huberman. Predicting the future with social media. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 1, pages 492–499. IEEE, 2010.

37 A. Bar-Or, J. Healey, L. Kontothanassis, and J. Van Thong. Biostream: a system architecture for real-time processing of physiological signals. In *Engineering in Medicine and Biology Society, 2004. IEMBS '04. 26th Annual International Conference of the IEEE*, volume 2, pages 3101–3104, Sept 2004. doi: 10.1109/IEMBS.2004.1403876.

38 B. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988. ISSN 0018-9162. doi: 10.1109/2.59.

39 W. Bosl, A. Tierney, H. Tager-Flusberg, and C. Nelson. Eeg complexity as a biomarker for autism spectrum disorder risk. *BMC medicine*, 9(1):1, 2011.

40 S. Chokroverty. *Sleep Disorders Medicine: Basic Science, Technical Considerations, and Clinical Aspects*. Elsevier Science, 2013. ISBN 9781483165196. URL https://books.google.co.in/books?id=tEAfAwAAQBAJ.

41 G. Clifford, J. Fisher, J. Greenberg, and W. Wells. HST.582J Biomedical Signal and Image Processing, Spring 2007. URL http://ocw.mit.edu/courses/health-sciences-and-technology/hst-582j-biomedical-signal-and-image-processing-spring-2007.

42 W. Commons. Epileptic spike and wave discharges monitored with EEG, 2006. URL https://commons.wikimedia.org/wiki/File:Spike-waves.png. File: Spike-waves.png.

43 W. Commons. ECG of a heart in normal sinus rhythm, 2007. URL https://en.wikipedia.org/wiki/Electrocardiography#/media/File:SinusRhythmLabels.svg. File: SinusRhythmLabels.svg.

44 R. G. Cooper. Stage-gate systems: a new tool for managing new products. *Business horizons*, 33(3):44–54, 1990.

45 B. Delgutte. Introduction to biomedical signal and image processing, Dec 2015. URL http://ocw.mit.edu/courses/health-sciences-and-technology/hst-582j-biomedical-signal-and-image-processing-spring-2007/lecture-notes/l1_intro.pdf.

46 A. Delorme and S. Makeig. EEGLAB: an open source toolbox for analysis of single-trial {EEG} dynamics including independent component analysis. *Journal of Neuroscience Methods*, 134(1):9 – 21, 2004. ISSN 0165-0270. doi: http://dx.doi.org/10.1016/j.jneumeth.2003.10.009. URL http://www.sciencedirect.com/science/article/pii/S0165027003003479.

47 A. Delorme, C. Kothe, A. Vankov, N. Bigdely-Shamlo, R. Oostenveld, T. O. Zander, and S. Makeig. Matlab-based tools for bci research. In *Brain-Computer Interfaces*, pages 241–259. Springer, 2010.

48 L. A. Farwell. Brain fingerprinting: a comprehensive tutorial review of detection of concealed information with event-related brain potentials. *Cognitive neurodynamics*, 6(2):115–154, 2012.

49 A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

50 N. F. Güler and E. D. Übeyli. Theory and applications of biotelemetry. *J. Med. Syst.*, 26(2):159–178, Apr. 2002. ISSN 0148-5598. doi: 10.1023/A:1014862027454. URL http://dx.doi.org/10.1023/A:1014862027454.

51 M. Guller. *Big Data Analytics with Spark: A Practitioner's Guide to Using Spark for Large Scale Data Analysis*. Apress, 2015. ISBN 9781484209646. URL https://books.google.co.in/books?id=yqhPCwAAQBAJ.

52 P. Hu, S. M. Galvagno, A. Sen, R. Dutton, S. Jordan, D. Floccare, C. Handley, S. Shackelford, J. Pasley, C. Mackenzie, et al. Identification of dynamic prehospital changes with continuous vital signs acquisition. *Air medical journal*, 33(1):27–33, 2014.

53 I. Iturrate, J. Antelis, and J. Minguez. Synchronous eeg brain-actuated wheelchair with automated navigation. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 2318–2325, May 2009. doi: 10.1109/ROBOT.2009.5152580.

54 D. Kan and P. Lee. Decrease alpha waves in depression: An electroencephalogram (eeg) study. In *BioSignal Analysis, Processing and Systems (ICBAPS), 2015 International Conference on*, pages 156–161. IEEE, 2015.

55 C. Koeninger, D. Liu, and T. Das. Improvements to Kafka integration of Spark Streaming, Oct 2015. URL https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html.

56 C. A. Kothe and S. Makeig. BCILAB: a platform for brain-computer interface development. *Journal of Neural Engineering*, 10(5):056014, 2013. URL http://stacks.iop.org/1741-2552/10/i=5/a=056014.

57 J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

58 A. Kumiega and B. Van Vliet. A software development methodology for research and prototyping in financial markets. *Quality Financial Management*, 7, 2006.

59 Y. Liu, O. Sourina, and M. K. Nguyen. Real-time eeg-based human emotion recognition and visualization. In *Cyberworlds (CW), 2010 International Conference on*, pages 262–269, Oct 2010. doi: 10.1109/CW.2010.37.

60 M. Manoochehri. *Data Just Right: Introduction to Large-Scale Data & Analytics*. Addison-Wesley Data & Analytics Series. Pearson Education, 2013. ISBN 9780133359077. URL `https://books.google.co.in/books?id=kmlCAgAAQBAJ`.

61 D. Mao, Y. Wang, and Q. Wu. A new approach for physiological time series. *CoRR*, abs/1504.06274, 2015. URL `http://arxiv.org/abs/1504.06274`.

62 A. A. Moenssens. Brain fingerprinting-can it be used to detect the innocence of persons charged with a crime. *UMKC L. Rev.*, 70:891, 2001.

63 G. D. F. Morales. Distributed stream processing showdown: S4 vs Storm, Oct 2015. URL `http://gdfm.me/2013/01/02/distributed-stream-processing-showdown-s4-vs-storm/`.

64 L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4257-7. doi: 10.1109/ICDMW.2010.172. URL `http://dx.doi.org/10.1109/ICDMW.2010.172`.

65 E. Niedermeyer and F. da Silva. *Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*. LWW Doody's all reviewed collection. Lippincott Williams & Wilkins, 2005. ISBN 9780781751261. URL `https://books.google.fi/books?id=tndqYGPHQdEC`.

66 P. Pandian, A. K. Whitchurch, J. K. Abraham, H. B. Baskey, J. Radhakrishnan, V. K. Varadan, V. Padaki, K. B. Rao, and R. Harbaugh. Low noise multi-channel biopotential wireless data acquisition system for dry electrodes. In *The 15th International Symposium on: Smart Structures and Materials & Nondestructive Evaluation and Health Monitoring*, pages 69310Q–69310Q. International Society for Optics and Photonics, 2008.

67 C. Prosser. *Comparative Animal Physiology, Environmental and Metabolic Animal Physiology.* Comparative Animal Physiology. Wiley, 1991. ISBN 9780471857679. URL https://books.google.co.in/books?id=7fQvbFlQBaQC.

68 T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 262–270, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1462-6. doi: 10.1145/2339530.2339576. URL http://doi.acm.org/10.1145/2339530.2339576.

69 T. M. Rath and R. Manmatha. Lower-bounding of dynamic time warping distances for multivariate time series. 2003.

70 S. Reiterer, E. Pereda, J. Bhattacharya, et al. On a possible relationship between linguistic expertise and eeg gamma band phase synchrony. *Frontiers in psychology*, 2(334):1–11, 2011.

71 Y. Renard, F. Lotte, G. Gibert, M. Congedo, E. Maby, V. Delannoy, O. Bertrand, and A. Lécuyer. Openvibe: An open-source software platform to design, test, and use brain–computer interfaces in real and virtual environments. *Presence: Teleoper. Virtual Environ.*, 19(1):35–53, Feb. 2010. ISSN 1054-7460. doi: 10.1162/pres.19.1.35. URL http://dx.doi.org/10.1162/pres.19.1.35.

72 W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0. URL http://dl.acm.org/citation.cfm?id=41765.41801.

73 S. Sanei and J. Chambers. *EEG Signal Processing.* Wiley, 2013. ISBN 9781118691236. URL https://books.google.fi/books?id=f44hLefOz6UC.

74 G. Schalk, D. McFarland, T. Hinterberger, N. Birbaumer, and J. Wolpaw. BCI2000: a general-purpose brain-computer interface (BCI) system. *Biomedical Engineering, IEEE Transactions on*, 51(6):1034–1043, June 2004. ISSN 0018-9294. doi: 10.1109/TBME.2004.827072.

75 I. Simanova, M. Van Gerven, R. Oostenveld, and P. Hagoort. Identifying object categories from event-related eeg: toward decoding of conceptual representations. *PloS one*, 5(12):e14465, 2010.

76 Y. N. Singh, S. K. Singh, and A. K. Ray. Bioelectrical signals as emerging biometrics: Issues and challenges. *ISRN signal processing*, 2012, 2012.

77 S. Smith. Eeg in the diagnosis, classification, and management of patients with epilepsy. *Journal of Neurology, Neurosurgery & Psychiatry*, 76(suppl 2):ii2–ii7, 2005.

78 S. Sriparasa. *JavaScript and JSON Essentials*. Community experience distilled. Packt Publishing, 2013. ISBN 9781783286041. URL `https://books.google.co.in/books?id=MZOkAQAAQBAJ`.

79 J. Sun and C. K. Reddy. Big data analytics for healthcare. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1525–1525, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2174-7. doi: 10.1145/2487575.2506178. URL `http://doi.acm.org/10.1145/2487575.2506178`.

80 M. Z. Tathagata Das and P. Wendell. Diving into Spark Streamingś Execution Model, Aug 2015. URL `https://databricks.com/blog/2015/07/30/diving-into-spark-streamings-execution-model.html`.

81 M. Teplan. Fundamentals of EEG measurement. *Measurement Science Review*, 2, 2012. URL `http://www.edumed.org.br/cursos/neurociencia/MethodsEEGMeasurement.pdf`.

82 C. Wasson. *System Engineering Analysis, Design, and Development: Concepts, Principles, and Practices*. Wiley Series in Systems Engineering and Management. Wiley, 2015. ISBN 9781118967140. URL `https://books.google.fi/books?id=wuJbCwAAQBAJ`.

83 M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

84 M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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.

85 M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

86 M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

87 Y. Zhang. Real-time development of patient-specific alarm algorithms for critical care. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 4351–4354. IEEE, 2007.

Appendix 1. DTW Algorithms

Algorithm 1 DTW naive algorithm

```
1: procedure getDtwDistanceNaive(channelNameRDDs, queryRDD)
2:     state1 ← Map(String, Array); state2 ← Map(String, Array)
3:     channelNameRDDs.collect().foreach{ch ⇒
4:     state1+=(ch → selfRDD.map(s ⇒ s.channels(ch)).collect())
5:     state2+=(ch → queryRDD.map(s ⇒ s.channels(ch)).collect())}
6:     return channelNameRDDs.map{channel ⇒
         series1 ← state1.get(channel); series2 ← state2.get(channel)
         (channel, computeNaiveDTW(series1, series2))}
7: end procedure
1: procedure computeNaiveDTW(seriesDV1, seriesDV2)
2:     dm ← Matrix(seriesDV1.size, seriesDV2.size)
3:     for i ← 0 until seriesDV1.size do
4:         for j ← 0 until seriesDV2.size do
5:             dist ← pow((seriesDV1(i) − seriesDV2(j)), 2.0)
6:             minValue ← min(min(MatrixValue(dm, i − 1, j),
               MatrixValue(dm, i, j − 1)), MatrixValue(dm, i − 1, j − 1))
7:             dm(i, j) ← dist + minValue
8:     return sqrt( dm(seriesDV1.size-1, seriesDV2.size-1))
9: end procedure
```

Algorithm 2 DTW algorithm using locality constraint optimization

1: **procedure** GETDTWDISTANCELC(channelNameRDDs,queryRDD,window)
2: $state1 \leftarrow Map(String, Array)$; $state2 \leftarrow Map(String, Array)$
3: $channelNameRDDs.collect().foreach\{ch \Rightarrow$
4: $state1 += (ch \rightarrow selfRDD.map(s \Rightarrow s.channels(ch)).collect())$
5: $state2 += (ch \rightarrow queryRDD.map(s \Rightarrow s.channels(ch)).collect())\}$
6: **return** $channelNameRDDs.map\{channel \Rightarrow$
$series1 \leftarrow state1.get(channel)$; $series2 \leftarrow state2.get(channel)$
$(channel, \text{COMPUTELCDTW}(series1, series2, window))\}$
7: **end procedure**

1: **procedure** COMPUTELCDTW(seriesDV1, seriesDV2, window)
2: $dm \leftarrow Matrix(seriesDV1.size, seriesDV2.size)$
3: $w \leftarrow scala.math.max(window, (s1.size - s2.size))$
4: **for** $i \leftarrow 0$ until seriesDV1.size **do**
5: **for** $j \leftarrow$ max(0, i-w) until min(seriesDV2.size, i+w) **do**
6: $dist \leftarrow pow((seriesDV1(i) - seriesDV2(j)), 2.0)$
7: $minValue \leftarrow min(min(\text{MATRIXVALUE}(dm, i-1, j),$
$\text{MATRIXVALUE}(dm, i, j-1)), \text{MATRIXVALUE}(dm, i-1, j-1))$
8: $dm(i, j) \leftarrow dist + minValue$
9: **return** sqrt(dm(seriesDV1.size-1, seriesDV2.size-1))
10: **end procedure**

Algorithm 3 DTW algorithm using LB_Keogh optimization

```
1: procedure getDtwDistanceLBKeogh(channelNameRDDs,queryRDD,radius)
2:     state1 ← Map(String, Array); state2 ← Map(String, Array)
3:     channelNameRDDs.collect().foreach{ch ⇒
4:     state1+=(ch → selfRDD.map(s ⇒ s.channels(ch)).collect())
5:     state2+=(ch → queryRDD.map(s ⇒ s.channels(ch)).collect())}
6:     return channelNameRDDs.map{channel ⇒
        series1 ← state1.get(channel); series2 ← state2.get(channel)
        (channel, computeLBKeoghDTW(series1, series2, radius))}
7: end procedure

1: procedure computeLBKeoghDTW(s1,s2,searchRadius)
2:     r ← searchRadius; LBSum ← 0.0
3:     for index ← 0 until seriesDV1.size do
4:         if (index − r) ≥ 0 then
5:             s2StartIndex ← (index − r)
6:         else
7:             s2StartIndex ← 0
8:         if (index + r) ≥ seriesDV2.size then
9:             s2StopIndex ← seriesDV2.size
10:        else
11:            s2StopIndex ← (index + r)
12:        lowerBound ← min(seriesDV2.slice(s2StartIndex, s2StopIndex))
13:        upperBound ← max(seriesDV2.slice(s2StartIndex, s2StopIndex))
14:        if seriesDV1(index) > upperBound then
15:            LBSum ← LBSum + pow((seriesDV1(index) − upperBound), 2)
16:        else
17:            LBSum ← LBSum + pow((seriesDV1(index) − lowerBound), 2)
18:    return sqrt(LBSum)
19: end procedure
```

Appendix 2. Seminar Paper

Analysis of systems to process massive data stream

Author: Maninder Pal Singh (maninder.singh@cs.helsinki.fi)

Seminar: Distributed Computing Frameworks for Big Data

Course Supervisor: Professor Sasu Tarkoma

Course Date: Sept, 2014 - Dec, 2014

State: Unpublished

Analysis of systems to process massive data stream

Maninder Pal Singh
maninder.singh@cs.helsinki.fi

ABSTRACT

The immense growth of data demands switching from traditional data processing solutions to systems, which can process a continuous stream of real time data. Various applications employ stream processing systems to provide solutions to emerging Big Data problems. Open-source solutions such as Storm, Spark Streaming and S4 are an attempt to answer key stream processing questions. The recent introduction of real time stream processing commercial solutions such as Amazon Kinesis, IBM Infosphere Stream reflects industry requirements. The system and application based challenges to handle massive stream of real time data are an active field of research.

In this paper, we present a comparative analysis of the existing state-of-the-art stream processing solutions. We also include various domains which are transforming their business model to benefits from stream processing.

Keywords

Stream computing, Massive data stream, Real time analysis, Streaming solutions

1. INTRODUCTION

The growth of massive data domains such as social networks, high frequency trading, online gaming, advertisement, DNA sequencing are beyond the reach of traditional data processing systems. Companies are focusing towards real time data based products, for the consumers. For example, the online gaming company such as Supercell[1] provides online games - Clash of Clans[2] and Boom Beach[3]. These games are online combat strategy games which can be played on devices such as tablets and smartphones. Supercell is using Amazon Kinesis [2] for real time processing of data streams generated from various devices. Amazon Kinesis helps in real time analysis of game insight data originated from hundreds of game engine servers [4]. The timely insight data helps in business analytics and is used to improve the game experience of the players [4].

The stream processing concept has evolved from stream computing paradigm which involves continuous query and real time analysis of massive stream of data. There are various solutions which address real time stream processing. S4 [19], Storm [7] and Spark Streaming [5] are examples of existing open-source solutions. Commercial solutions such as Amazon Kinesis [2], IBM Infosphere stream [21] are also working in the direction of stream processing.

The paper focuses on discussion and comparative analysis of existing state-of-the-art stream processing solutions. The rest of the paper is organized as follows. Section two discusses about massive data streaming concepts. Section three looks into various streaming solutions. Section four discusses the architecture perspective of stream processing solutions and explores programming model, latency, data pipelines, fault tolerance and data. Section five presents emerging use case of stream processing solutions. Section six explores challenges of stream processing solutions and applications. Finally, the conclusions are summarized in section seven.

2. STREAMING CONCEPTS

Streaming data is fundamentally different from traditional data handling patterns and comes with its own set of challenges and requirements. It requires in-stream processing to have a low latency. The system should be scalable with self load balancing capability and should have high availability. It may require some persistent storage for short period of time. Real time streaming data has all the well-known attributes of Big Data, such as volume, variety, velocity and veracity.

Stream processing requires handling of varying rate of streaming data, the incoming streaming data might involve missing data or delays. The processing includes on-the-fly decision and provision for handling such out of order streaming data. The streaming data can be time-stamped on arrival or subject to discard depending upon sensitiveness of data. The time sensitive operations require time-out of blocking computations. The time intensive operations require careful handling of stream linked to the system and resources binding to the computation. Real

[1]http://www.supercell.net/
[2]http://www.supercell.net/games/view/clash-of-clans
[3]http://www.supercell.net/games/view/boom-beach

Table 1: Open Source Streaming Solutions

Solution	Type	Developed By
Storm	Streaming	BackType
Spark Streaming	Batch & Streaming	Berkeley AMPLab
S4	Streaming	Yahoo

Table 2: Commercial Streaming Solutions

Solution	Developed By
MillWheel	Google
Amazon Kinesis	Amazon
Infosphere	IBM

time streaming also requires deterministic processing as time-order guarantee is subjected to conditions. The system demands mining around processed streaming data and data stored across persistent storage. The persistent storage adds additional latency, but requires integration to provide business analytic around data. Persistent storage for long time of streaming data involves its own set of challenges. The streaming data require extensible framework for querying and processing to conclude desired results. The operators involving stream data require understanding of streaming data context. The variety of data in the structured, unstructured or semi-structured form requires adaptability in real time processing.

The attributes such as data integrity and data availability are an integral part of data engines. There is a change in paradigm which involves distributed processing solutions provider to run on low cost commodity hardware clusters. The system demands scalability and transparent load-balancing for such high volume of data. The data engine should be adaptive, extensible to add easy to program modules and capable to process high-volume of streaming data with low latency.

3. STREAMING SOLUTIONS

There are few solutions available for real time massive data stream processing. The available solutions can be classified into open source contributions and commercial solutions. The Table 1 refers to various open-source solutions available for real time massive data streaming processing. Table 2 refers to the list of various streaming solutions employed by industry.

3.1 Open Source Streaming Solutions

Storm is a distributed stream processing framework, developed in Clojure and built upon model of task parallel computation [7]. It provides an adapter to write applications in virtually any language. Storm is optimized for low-latency processing and uses ZeroMQ[4] for message passing, which makes its architecture to provide a guaranteed message processing [7]. It attempts to process each record at least once and if a record is not yet processed by a node, it replays the records. In addition, It provides fair fault detection and process management. On discovery of failure of a task, messages are automatically reassigned by quickly restart the processing. For optimal resource handling, the processes in Storm are managed by supervisors.

Spark Streaming is an extension of the core Spark API and an in-memory distributed data analysis platform [5]. Spark is built upon the model of data parallel computation. It provides reliable processing of live streaming data. Spark streaming transforms streaming computation into a series of deterministic micro-batch computations, which are then executed using Spark's distributed processing framework.

S4 (Simple Scalable Streaming System) is a general purpose distributed and scalable streaming platform that allows the processing of continuous unbounded streams of data. Its processing model is inspired by MapReduce and uses key based programming model [18]. The computation is performed by processing elements and messages are transmitted between them in the form of data events [19].

3.2 Commercial Streaming Solutions

Google[5] **MillWheel** is a framework for low-latency data processing streaming applications [1]. It is also inspired by MapReduce programming model and allows users to write application logic in a directed computer graph using custom topology [1]. Records in a Google MillWheel are delivered continuously along edges in a graph [1]. It provides fault tolerance and guaranteed delivery of records to the users.

Amazon[6] **Kinesis** is a service to perform real time processing of massive data stream [3]. It is a recent solution which was introduced in late 2013. Kinesis adapts to streaming data and do load-balancing by auto-scaling. Fault tolerance is provided using checkpointing to replay data records. Kinesis comes with a Kinesis client library that requires users to create "Producer" and "Worker" in an application. The Producer accepts data from stream source and converts it into a Kinesis stream. Kinesis stream consists of data records organized into tuples. The Worker acts as client application which accept Kinesis stream and perform required processing on stream. The worker can be invoked on stream of data to obtain required results. The processed data is available only for 24 hours, which requires user to link storage solution for future processing.

IBM[7] **InfoSphere Streams** (Streams) is high-performance stream processing system [8]. It is used for large scale continuous real time in-stream data processing [8]. The Streams does not follow specific programming model. The Stream Processing Language (SPL) has been used for stream applications. SPL is a declarative programming language [8]. SPL allows users to create complex applications without focussing on intricacies of distributed execution [8]. Users can control operator and its execution using C++ or Java.

Streams includes various management services which together lays the foundation of distributed execution. Stream application accepts jobs and performs concurrent

[4]http://zeromq.org/

[5]https://www.google.com/

[6]http://www.amazon.com/

[7]http://www.ibm.com/

Table 3: Attributes based Streaming solution classification

Attributes	Storm	Spark Streaming	S4
Framework	Stream Processing + Micro-Batching using Trident	Micro-Batching with Batch Processing using Core Spark	Actor Programming Model
Implemented in	Clojure	Scala	Java
Application Language	Java, Clojure, Scala, Python, Ruby	Java, Scala	Java, Python, C++
Stream Primitive	Tuples	DStream	Events
Stream Source	Spouts	Network, HDFS	Network
Computation or Transformation	Bolts	Transformation, Window Operations	Processing Element
Persistence Entity	Bolts	foreach RDD	Control Messages
Reliable Execution	At least once	Exactly once	–
Fault Tolerance	Tuples replayed, Guaranteed delivery	Tiny bits loss possible, Require HDFS for fully fault tolerant	New Node begin from snapshot
Latency	Sub-Second	Few Seconds	Few Seconds[a]
Developed By	Conceived by BackType/ Twitter, Now Apache incubation project	Conceived by AmpLab Berkely, Now Apache incubation project	Initially conceived by Yahoo!, Now Apache incubation project

[a]Assuming low latency as few seconds [19]

processing. A job consists of one or more Processing Elements (PEs) [8]. Messaging in the system is performed using Low Latency Messaging (LLM) mechanism to optimize application execution. IBM Infosphere Streams is actively used in diverse domains such as transportation [9], DNA sequencing [17], radio astronomy [10], weather forecasting [13], stock market trading [6], and telecommunications [16].

4. ARCHITECTURE ANALYSIS

The streaming concept has been divided into micro-batching processing technique or non-batch processing techniques. Spark Streaming solution provides micro-batching of unbounded stream. It incorporates stream processing via short interval of batches and provides linear streaming solution which is suitable for existing batch processing infrastructure. Storm and S4 both adopted non-batch processing techniques. Storm also provides micro-batch processing using Trident APIs. Apache S4 is entirely focussed on real time stream processing and does not support micro-batch processing.

The attribute based comparison is performed in Table 3 between Storm, Spark Streaming, and S4. Table 3 highlights different aspects of these solutions, which can be compared in context of processing model, data pipeline, latency, fault tolerance, and data guarantees.

4.1 Processing Model & Latency

Storm does not mandate any specific programming model. It adopted both streaming processing and complex event processing [20]. It follows Directed acyclic graph (DAG) as a processing model. DAG is a directed graph with no directed cycles. Storm provides topologies that operate on stream of data. A topology is a job and is represented as DAG. The vertex in a topology represents a worker and edges represent the dataflow between the worker instances. Workers are classified as spouts and bolts. Therefore, as topologies are arranged in a DAG, the data flows from spout to bolt and reverse flow is not possible. Spouts work as an input source for the topology. Since, incoming events are processed as one record at a time, Storm have sub-second latency.

Spark streaming follows a micro-batching programming model. It combines streaming model with batch processing model. Before processing arrived data, Spark streaming batches up events within a short time frame. The batch processing of smallest units in Spark streaming leads to few second latency.

S4 implements the Actors programming paradigm [18]. Processing elements are defined by the user. The messages as data events are transmitted between processing elements [19]. The triggered events are consumed by the S4 processing elements. Processing Elements interact with each other either as an event generator or event consumer. S4 is inspired from MapReduce model.

4.2 Data pipeline

Storm employs pull model where events from sources are pulled by each bolt. The loss of events is possible only at ingestion time. Spouts are responsible for maintaining the event rate.

On the other hand, Spark follows micro-batching processing model where massive data streams are divided in small batches and considered as Resilient Distributed Dataset (RDD). RDD is a distributed memory abstraction that

allows in-memory computations on large clusters in a fault-tolerant manner [24]. RDD is smallest processing unit and results of RDD operations are returned as batches.

Finally, S4 is based on push model. The data events are pushed to appropriate Processing Elements. There is a possibility of drop of data events in case of choking of receiver buffer.

4.3 Fault Tolerance & Data Guarantees

As Storm processing model is based on a record, therefore, state of each record requires to be tracked as arrived in DAG nodes. Storm only guarantees processing of record to be atleast once. In case of failure, the records can be replayed by spout. It is quite possible to have duplicates or twice updates to the mutable state of record. Events are possible to be lost due to various reasons, therefore, the state recovery is important from system perspective. State recovery is also one of the required attribute for long running operations. Storm does not provide state recovery but provides gauranteed delivery and processing of data.

Spark Streaming and Storm both provide fault tolerance and data guarantees. Stateful computation is better supported in Spark Streaming. Spark Streaming guarantees that batch level processing will be executed in an exactly one manner. In case of a node failure, Spark Streaming allows rebuilding a dataset in a node.

S4 provides state recovery via uncoordinated checkpointing [12]. In case of failure or crash, the other nodes begin operation with recent snapshot of its state. The data events to the Processing Elements may be sent with or without guaranteed delivery. S4 also provides gauranteed delivery of control messages.

5. APPLICATIONS

The rise of various solutions to process real time continuous stream of data reflects the trends and interest of public in massive data stream. The stream processing systems are adopted by variety of applications from social media to sensing devices to astronomical telescope. An overview of such applications are provided below.

Finance services are based on high frequency real time trading and investment. Most of the transactions are performed using credit cards by customers. Banking institution has to take preventive measures to detect any fraud activities with credit cards [23]. For that purpose, banking sector monitors and processes multiple streams of transaction every day. The real time monitoring of transactions prevents likelihood of miscellaneous activities. Real time stream processing system plays an important role in decision making for trading and investment.

Medical hospitals are also involved in using distributed stream processing for health monitoring objectives. The streams of measurement data generated from various medical instruments are processed and analyzed for proactive health diagnosis. The real time stream based monitoring tool assists doctors for diagnosis and relieves workload from other staff of hospital [11].

Smart cities [15] explore urban planning to incubate human adaptive environment. The real time data from different domains is analyzed for city planning and human mobility [14]. The urban data from cities are explored to assist government in dynamic decision making process [22]. These distributed real time streams of data can be used for optimization of public transportation. It also allows people to avoid traffic congestion across different routes within a city. The urban data can also be used for constant weather monitoring and air content monitoring.

Radio Astronomy involves continuous stream of data from radio telescopes. The telemetry communication process collects continuous stream of data remotely using various radio elements such as antennas, beam formers. These imaging signals are synthesized and processed at real time. The final accumulated outcome is stored in a system. There are number of projects which are utilizing streaming solutions such radio astronomy group of Uppsala University and the LOFAR Outrigger In Scandinavia (LOIS) project [10] [21].

DNA sequence analysis requires large-scale data set processing along with incremental computation and parallel processing while handling linear scalability. The Next-Generation Sequencing (NGS) methods benefit from streaming data analysis in a scalable and cost-efficient way. The stream computing provides promising solution for large scale data-intensive computations in domain such as bioinformatics. The stream-based data management solution for large-scale DNA sequence analysis is explored using IBM Infosphere Streams computing platform [17].

There are endless possibilities to utilize real time streaming data. Various fields such personalization of web page by Yahoo![8], pay-as-you-drive insurance model, recommendation system, weather forecasting, energy trading services are emerging domains which are transforming their business model to have benefits from real time data stream processing. With the major development in Internet of Things, distributed real time stream processing and analysis soon will be the part of every day life.

6. CHALLENGES

The stream processing solutions are designed to solve emerging Big Data trends. The solutions and applications incorporate their own set of challenges, which should be addressed before designing any solution. The challenges require elaborate reasoning and inspection of application requirements to create an optimal solution. However, the challenges can be classified into application challenges and system level challenges.

6.1 Application Challenges

There are many domains which incorporated stream processing into applications. Each application is having its own set of requirements which provide uniqueness to them. Table 4 provides an overview of application challenges from domains such as radio astronomy imaging, smart cities, online gaming, medical hospitals, financial services include

[8]https://www.yahoo.com/

Table 4: Applications using streaming solution in real time environment

Applications	Applied Streaming Solution	Challenges
Online Gaming (esp. Supercell) [4]	Amazon Kinesis	• real time data streams originated from multiple players • continuous query on data streams to improve player experience • real time player sessions to provide real time experience • business analytics on real time insight of game data
Medical Hospitals [11]	IBM Infosphere Stream	• privacy-protected real time stream monitoring from medical devices • analysis of data streams to explore correlation in patient diseases • predictive proactive medical alerts from real time data streams • handling multiple data streams on large scale from multiple patients
Radio Astronomy Imaging [10]	IBM Infosphere Stream	• large volume of imaging data from multiple channels • handling of high incoming data rate • real time image synthesis for analysis • storage limitation as all raw data is not useful
DNA Sequencing [17]	IBM Infosphere Stream	• large volume of genetic data • large-scale DNA sequence analysis • high latency and significant processing time • incremental and parallel processing
Smart Cities [9]	IBM Infosphere Stream	• large volume of raw data from various source in cities • data disparity due to unstructured and unrelated raw data • modeling of heterogeneous data and real time data analogy
Finance Services [23] [6]	Storm, IBM Infosphere Stream	• real time decision on investing and trading • analytics around real data stream and previous stored market data • monitoring of millions of high frequency transactions • sub-second latency

data handling challenges. The high volume of data leads to high latency in DNA sequencing. The modeling of heterogeneous data and real time data analogy is a challenge for smart cities. The real time analysis of business analytics data is an important requirement for financial services. The adaptive real time experience for players in online gaming requires continuous query on real time data. The solutions require low latency for these domains to adapt with real time stream of massive data.

6.2 System Challenges

The stream processing system encounters following challenges which can be broadly categorized into four categories.

Data Acquisition: It is challenging to handle massive stream of continuous data. The system requires to adapt to the velocity of incoming data. The variety of incoming data described as structured or unstructured data. The structured data acts as an optimal input for stream processing systems, whereas the unstructured data requires data preprocessing which involves filtering, extraction and organization into structured format. The latency of the stream processing system varies with structured and unstructured data. The correct representation of data and data acquisition strategies depend on the application built on the top of stream processing systems.

Data Handling: Another challenge is to properly handle large volume of data. The stream processing application requires to analyze sensitivity of data which need to store into persistent storage. Some applications only requires to store cumulative processed results while other applications require to store filtered and structurally organized processed data for later usage and analysis. The data handling and persistent storage of data format varies with the application requirement. It needs to be properly assessed by stream processing systems.

Data Modeling: The stream processing systems require in-stream processing capabilities to have a low latency. Considering the volume, variety, velocity and veracity of data, the stream processing system requires predictive models and efficient algorithms to extract application linked to important events from massive data streams. It also requires to have data models to perform comprehensive analysis by combining all available data.

Data Mining: The stream computing involves computational analysis and analytics around it. The stream processing requires new computational tools which can analyze heterogeneous data sets into appropriate results. It involves data analytics and data visualization of

massive data sets. The traditional mining approaches need to adapt as per in-stream processing to provide dynamic results.

7. CONCLUSIONS

In the last decade, significant research has been performed to create a system that can handle Big Data. The MapReduce paradigm is able to offer a solution for Big Data and many solutions revolves around it. A solution based on MapReduce is suitable for many problems but not appropriate for many others. Previous research has been paired to find solutions which would be optimal for emerging Big Data trends. The stream computing paradigm appears to be a solution to the emerging Big Data trends.

The research community is primarily focused on development of solution or mining of large data sets. The research on providing solution is divided into the selection of the programming model or data model for a system. The selection of processing model for a system varies from batch processing to micro-batch processing. Considering the availability of MapReduce as successful paradigm, many solutions for streaming are influenced by this paradigm. Some solutions also explore the Actor model to have stream processing solution. Solutions such as Storm provide a sub-second latency and S4 does not provide persistent state and complete fault tolerance. Spark Streaming has mixed processing model and exactly once mechanism for record delivery which might affect processing.

A fundamental set of questions exists, which should be addressed before selecting any programming model or data model for stream processing. The design choices and challenges affect system latency and throughput. The challenges linked to applications and processing system require elaborate reasoning and inspection of requirements to create a stream processing system for heterogeneous data set. The stream processing paradigm requires solution which can provide low latency, high throughput, fault tolerance along with scalability and versatility. The system requires extensibility to plug and play different components to provide analytics for in-stream processing and stored stream data in a persistent storage.

8. ACKNOWLEDGEMENTS

I sincerely thank the reviewers for their comments and suggestions. This survey paper has been supported by the University of Helsinki as a learning initiative under Seminar course on distributed computing frameworks for Big Data.

9. REFERENCES

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[2] Amazon. Amazon Kinesis. `http://aws.amazon.com/kinesis/`.

[3] Amazon. Amazon Kinesis Product Details. `http://aws.amazon.com/kinesis/details/`.

[4] Amazon. AWS Case Study: Supercell. `http://aws.amazon.com/solutions/case-studies/supercell/`.

[5] U. B. AMPLab. Spark Streaming. `http://spark.incubator.apache.org/docs/latest/streaming-programming-guide.html`.

[6] H. Andrade, B. Gedik, K.-L. Wu, and P. Yu. Scale-up strategies for processing high-rate data streams in system s. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1375–1378, March 2009.

[7] Apache. Storm - Distributed and fault-tolerant realtime computation. `http://storm.apache.org/`.

[8] C. Ballard. *IBM Infosphere Streams harnessing data in motion*. Vervante, S.l, 2010.

[9] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1093–1104, New York, NY, USA, 2010. ACM.

[10] A. Biem, B. Elmegreen, O. Verscheure, D. Turaga, H. Andrade, and T. Cornwell. A streaming approach to radio astronomy imaging. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 1654–1657, March 2010.

[11] M. Blount, M. Ebling, J. Eklund, A. James, C. McGregor, N. Percival, K. Smith, and D. Sow. Real-time analysis for intensive care: Development and deployment of the artemis analytic system. *Engineering in Medicine and Biology Magazine, IEEE*, 29(2):110–118, March 2010.

[12] J. Chauhan, S. A. Chowdhury, and D. Makaroff. Performance evaluation of yahoo! s4: A first look. In *Proceedings of the 2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '12, pages 58–65, Washington, DC, USA, 2012. IEEE Computer Society.

[13] L. Daldorff. Novel data stream techniques for real time hf radio weather statistics and forecasting. *IET Conference Proceedings*, pages 74–76(2), January 2009.

[14] C. Harrison, B. Eckman, R. Hamilton, P. Hartswick, J. Kalagnanam, J. Paraszczak, and P. Williams. Foundations for smarter cities. *IBM Journal of Research and Development*, 54(4):1–16, July 2010.

[15] J. M. Hernández-Muñoz, J. B. Vercher, L. Muñoz, J. A. Galache, M. Presser, L. A. H. Gómez, and J. Pettersson. The future internet. chapter Smart Cities at the Forefront of the Future Internet, pages 447–462. Springer-Verlag, Berlin, Heidelberg, 2011.

[16] IBM Corporation. Exploiting Big Data in telecommunications to increase revenue, reduce customer churn and operating costs. `http://www-01.ibm.com/software/data/bigdata/industry-telco.html`.

[17] R. Kienzler, R. Bruggmann, A. Ranganathan, and N. Tatbul. Large-scale dna sequence analysis in the cloud: A stream-based approach. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 467–476, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] G. D. F. Morales. Distributed stream processing

showdown: S4 vs storm.

[19] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[20] J. Pillgram-Larsen. Storm - the Hadoop of stream processing. `http://fierydata.com/2012/03/29/storm-the-hadoop-of-stream-processing/`.

[21] R. Rea. IBM InfoSphere Streams. `http://www.monash.com/uploads/IBM-InfoSphere-Streams-White-Paper.pdf`.

[22] H. Schaffers, N. Komninos, M. Pallot, B. Trousse, M. Nilsson, and A. Oliveira. The future internet. chapter Smart Cities and the Future Internet: Towards Cooperation Frameworks for Open Innovation, pages 431–446. Springer-Verlag, Berlin, Heidelberg, 2011.

[23] L. Sensmeier. How Big Data is revolutionizing Fraud Detection in Financial Services. `http://hortonworks.com/blog/how-big-data-is-revolutionizing-fraud\-detection-in-financial-services/`.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.