university of groningen

faculty of mathematics and natural sciences

SMART: Real-time Analytics for Environment Monitoring & Actuation

Author: Mohamed ElSioufy

Master Thesis

Faculty of Mathematics & Natural Sciences
University of Groningen

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

August 2015

Abstract

Time-series data processing and real time data analysis are big issues nowadays. Recent advances in sensor technologies as well as their innovative designs allows them to be plugged anywhere, reporting thousands of measures each second. With the development of sensing, wireless communication, and Internet technologies, almost every object in our everyday life has the ability to emit data. How to store and query these enormous amounts of data efficiently remains an open research question. In this project we developed an extendable cloud-based system for the analysis and visualization of time-series data in real-time. We target a specific IoT setting, where a physical environment, embedded with sensors that continuously measure and report its state is monitored, analyzed and queried about its state-behaviours. The proposed system is built as an extensible software library on top of Apache Spark, a cluster computing engine and integrates with other technologies for IO and Visualization . We have applied and tested our solution on a simulated HVAC domain, where a set of rooms equipped with sensors have been monitored, analyzed and queried later to regulate their individual temperature levels.

Acknowledgments

First and foremost, I would like to thank God, the Almighty, for having made everything possible by giving me strength and courage to do this work.

I would like to express my sincere gratitude to my primary supervisor Prof. Alexander Lazovik for the continuous support with my thesis and related research; for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and I could not have imagined having a better adviser and mentor. I would also like to thank my thesis supervisors: Dr. Apostolos Ampatzoglou and Dr. Andrea Pagani, for their insightful comments, encouragement and trust. My sincere thanks for Dr. Andrea Pagani, who provided me his value-able feedback on my thesis manuscript.

Last but not the least, I would like to thank my family: my parents and my brother for supporting me spiritually throughout working on this thesis and my my life in general.

Contents

1 Introduction

The Internet of Things (IoT) is a concept that envisions all objects around us as part of the Internet. Every object is uniquely identified and is capable of sending information about its location, status and identity. The information and the related data are essential to characterize the environment, and the more data is available, the more opportunities arise to use such data to provide smart services and data-informed decisions.[69].

Over recent past years, there has been a growing interest in the ability of embedded devices, sensors and actuators to communicate. The recent progression in sensor manufacturing technologies together with advances in short range mobile communication and improved energy-efficiency allow sensors and embedded devices to be widely used in various IoT applications. Multi-

sensor systems can significantly contribute to the enhancement of the quality and availability of information. Thousands to millions of time-stamped data points are collected by sensors, smart meters, RFIDs and many more every second of every day. With the devices connected to the Internet, this huge amounts of data could be communicated and leveraged, providing a huge window of opportunities in various fields.

Applications that interact with devices like sensors have special requirements of massive storage to store big data, huge computation power to enable real time processing of the data, and high speed network for streaming the data in real time[84]. Cloud computing on the other hand has long been recognized as a paradigm for big data storage and analytics. Furthermore, it provides a perfect infrastructure to enable those real-time processing applications. Cloud platforms allow the sensing data to be stored and used intelligently for smart monitoring and actuation. Novel data fusion algorithms, machine learning methods, and artificial intelligence techniques can be implemented to run distributed on the cloud to achieve automated decision making[77].

In this project we apply our knowledge and research in the fields of cloud computing and big data providing a cloud-based, real-time data analytics and visualization platform for environmental monitoring and actuation. We assume an IoT setting where physical items are no longer disconnected from the virtual world, but can be accessed and controlled remotely. In our model, multiple physical environments equipped with sensors are monitored and analyzed in real-time. We provide a configurable system that continuously learns about defined behaviours of the target environments and could be used to provide predictions and visualizations about those behaviours. The system employs a form of predictive analytics and takes into account behavioural changes that could be accompanied with various forms of disruptions on the monitored environments e.g. seasonal climate change. Moreover, the system

is designed to benefit from data-sharing across multiple target environments, improving its prediction accuracy beyond the capabilities of "individual" things.

We leverage the power of cloud computing technologies to provide data scalability and rapid visualization, and we provide extensibility for user programmable analysis. We chose Apache Spark[11] as the underlying computing platform for our analysis. Spark is a large-scale data processing engine, and is associated with several high-level libraries including data-streaming capabilities and large-scale, distributed machine learning algorithms. Together with Spark, we use the combination of Apache Kafka[6], Rabbit-MQ[47], Apache Cassandra[4] and Lightning-viz[39] as complementary technologies to handle different aspects of the big data. The details and peculiarities of each of these technologies are described thoroughly in Chapter2 of this thesis.

Further, this thesis is organized as follows: In Chapter 2, we provide a technology overview. We present the technologies and frameworks that we have chosen to utilize and the rational behind our choices. In Chapter 3, we present and discuss the related work. In this chapter we zoom-into similar systems that has been proposed for similar kinds analytics. Chapters 4, 5 and 6 includes all the technical and theoretical aspects behind our system. We start in Chapter 4 by presenting a developed API for the domain of time-series. The API is an independent module that includes a set of common manipulation operators for time-series data and is utilized by the system in order to perform its functionality. In Chapter 5 we present the system. In this chapter we go through the system modeling as well as relevant technical details. We also present various concepts that we have considered during our design and implementation. In Chapter 6 we present different deployment strategies for the system. Finally, In Chapter 7, we conclude our work.

2

Technology Overview

Nowadays, we are experiencing rapidly increasing data availability because of the growing coverage of sensors, penetration of mobile Internet and the popularity of social media activities[79]. The types of data being created are likewise proliferating and the need to analyze the data in real-time to derive valuable information is increasing every day. To accommodate these needs, technologies are born to handle the huge datasets and overcome the limitations of previous products; providing new approaches for processing, storing and analyzing massive volumes of multi-structured data.

A quite extensive list of big data related technologies is provided in [80], including categories for distributed frameworks, programming models, data-stores, query engines, message oriented middlewares, large-scale machine

learning, data visualization and applications. While technologies classified under the same category could be thought of as alternatives, it is not the goal of this work to carry out a complete, comprehensive technology research; especially that many of these technologies have emerged outside the research environment. We however focused on researching technologies provided by big-data players, reputable organizations and institutes.

Our foundational technology requirement was a distributed stream-processing engine. In our model, we assume that the sensors data is streamed into the system at moderate latencies of minutes, thus near zero latency requirement of most developing real-time applications is not a major restriction. We however demand a high throughput, scalable stream-processing engine with adequate fault tolerance and durability. Our technology research has lead us to choosing Apache Spark, a general engine for large-scale data processing. We used Apache Spark together with Apache Kafka and Rabbit-MQ for IO, Apache Cassandra for data persistence and Lightning-viz for data visualization. In the following section, we present the rational behind our choice to Apache Spark among other exiting stream-processing engines. Subsequent sections describe the rational behind our choices to each of the remaining technologies. In these sections we do not focus extensively on comparisons between each technology and its alternatives, but highlight peculiar characteristics that lead to our particular choices.

2.1 Distributed Stream Processing

Stream processing is the in-memory, record-by-record analysis of data in motion. Data is typically in the form of unstructured log records or sensor events, with each record including a timestamp indicating the exact time of data creation or arrival. Typically, the objective is to extract actionable intelligence and react to operational exceptions through real-time alerts

and automated actions[59]. According to [88], three basic tenets distinguish stream processing engines from batch processing: (1) they must support primitives for streaming applications, (2) they must adopt a form of "inbound processing" where processing is performed directly on incoming data before (or instead of) storing them and (3) they must have capabilities to gracefully deal with spikes in data loads.

Along our technology research we have identified six, general purpose distributed stream processing engines. These are: S4 by Yahoo[81], TimesStream by Microsoft[83], Apache Samza[7], Apache Storm[87], Apache Flink[85] and Apache Spark[90]. We have also passed along other more limited stream processing engines like Puma from Facebook[50] and Apache Flume[5], which are bound to aggregation functionality, and Trident which is a high-level abstraction for real-time stream computing on top of Apache Storm. All of these frameworks have borrowed concepts from MapReduce[70] to separate programming logic from concerns of distributed systems.

Among the mentioned technologies we were interested in a general and open-source engine that has a considerable ecosystem and high-credibility in the research community. These were essential requirements next to scalability, fault tolerance, high throughput and durability. Our requirements allowed us to undoubtedly exclude the closed-source Microsoft and Yahoo engines as well as Apache Samza which does not provide a solid research background. We then had to chose among Apache Spark, Apache Flink and Apache Storm, and we have decided to chose Apache Spark. In the following sub sections we provide the reasons behind our particular choice to Apache Spark.

2.1.1 Apache Spark vs. Apache Storm

Apache Spark and Apache Storm are very popular engines that offer real-time processing capabilities to a wide class of users and applications. Both are

top-level projects within the Apache Software Foundation[9][10], and while the two tools provide overlapping capabilities, each have distinctive features and processing models.

Sparks employs a batch processing model that is based on RDDs or Resilient Distributed Datasets. RDDs are distributed memory abstractions, that are great for pipelining parallel operators for computation and are, by definition, immutable, which allows Spark to have a unique form of fault tolerance based on lineage information[89]. Spark Streaming is built on top of the RDDs model. It batches up events that arrive within a user-defined time window and process these events as RDDs, i.e. batches[90]. This micro-batching concept limits Sparks latency to seconds. Storm on the other side is designed for stream processing or what is called complex event processing[87]. Storm processes incoming events one at a time, providing fault tolerance for performing a computation or pipelining multiple computations on an event as it flows into a system. This allows storm to achieve sub-second latency of processing an event. The trade-off however is in the fault tolerance data guarantees. Spark Streaming provides exactly once processing guarantees for each record, thus better support for stateful computation that is fault tolerant. Storm guarantees that each record will be processed at least once, but allows duplicates to appear during recovery from a fault. That means mutable state may be incorrectly updated twice.

The different processing models allowed us to reason about different fault-tolerance strategies employed by each engine as well as their different latencies. We relied on the research provided in [79] to get acquainted with the relative throughputs of Spark and Storm. [79] is an initiative for bench-marking modern distributed stream computing frameworks. A streaming benchmark utility is provided that covers several micro workloads addressing typical stream computing scenarios and core operations. The bench-marking pro-

grams take into account the performance of the target streaming frameworks under different workloads as well as their durability and fault-tolerance. The authors have implemented their program-set on Apache Spark and Apache Storm, and concluded that Spark tends to have a larger throughput and less node failure impact compared to Storm, while Storm has much less latency except with complex workloads under large data scale for which its latency may be even multiple times of Spark; Spark and Storm demonstrated durability under constant workloads. The results by this research supported what we have mentioned earlier about latency and fault tolerance and was indeed in favour of Apache Spark, given that our system is designed for moderate latencies of minutes.

A final reason that complements our choice for Apache Spark over Storm is its unified programming model. While Storm has a special focus on stream processing, Sparks programming model covers batch and stream processing, and provides an integrated query engine, graph processing capabilities and distributed machine learning algorithms. All of these are implemented as high-level libraries on-top of Spark RDDs and can be seamlessly combined in the same application. The greatest advantage of this unified model is that its only one system to learn and manage.

2.1.2 Apache Spark vs. Apache Flink

Apache Flink is another very interesting platform for batch and stream processing. Similar to Spark, Flink is a top-level Apache project[8] and includes API's for query processing, machine learning and graph processing, thus alleviates the need to learn different programming paradigms when crafting an analysis. Flink on the other hand is a pure stream-processing engine and offers a fault tolerance mechanism with exactly once processing guarantees. The central part of Flink's fault tolerance mechanism is drawing consistent snapshots of the distributed data stream and operator state. These snapshots

act as consistent checkpoints to which the system can fall back in case of a failure. Flink's fault-tolerance is inspired by the standard Chandy-Lamport algorithm for distributed snapshots[67].

While Flink's processing model, fault tolerance and latency guarantees sounds very promising, it doesn't have the same prominence as Apache Spark or Apache Storm. At the time of our writing, the Spark-project has over 600 contributor[29] while Flink did not exceed 120[28]. We couldn't find any previous work that benchmarks Flink against Spark or Storm thus we do not have a clear research-backed up vision on Flink's performance as compared to any of Spark or Storm. However, Flink's continuous checkpointing for fault-tolerance could be very deleterious for performance.

We have decided to chose Spark over Flink because of its larger research community, user-base and active deployments. Moreover, Spark has beaten the world record for sorting data on-disk in 2014, 3x faster and using 10x fewer resources than MapReduce the previous record holder[51]. Finally, it is important to mention that for Flink's fault tolerance mechanism to realize its full processing guarantees, the data stream source, such as message queue or broker, needs to be able to rewind the stream to a defined recent point. This adds a limitation to Flink's streaming sources[26].

2.1.3 Conclusion

Storm and Flink could be alternatives to Spark if real-time data streaming is required with stringent latencies that Spark's micro-batch processing cannot provide. The benefit of Spark's micro-batch model on the other-hand is that we get full fault-tolerance and "exactly-once" processing for the entire computation, meaning it can recover all state and results even if a node crashes. Flink and Storm don't provide this, requiring application developers to worry about missing data or to treat the streaming results as potentially

incorrect.

2.2 Message Oriented Middleware

Messaging enables software applications to connect and scale. Applications can connect to each other as components of a larger application or to user devices and data. Our streaming platform continuously performs real-time processing and transformation on time-series state data arriving from sensors and queries arriving from user-applications. To reliably bring those data-streams into our platform, a messaging queue is essential. Spark Streaming provides out of the box connectivity for various streaming sources, including Apache Kafka, Apache Flume, Twitter, ZeroMQ, Kinesis and Raw TCP. We have chosen to support Apache Kafka and Rabbit-MQ as our streaming sources. In the following sub-sections we highlight the features of each of these systems and the reasons behind our choices.

2.2.1 Apache Kafka

Apache Kafka is an open source, distributed commit log service, that provides the functionality of a messaging system with a very unique design. It was originally developed by LinkedIn, and was subsequently open sourced in early 2011. Kafka aims to provide a unified, high-throughput, low-latency platform and is very convenient for handling real-time data feeds[76].

Overview

Kafka is designed very differently than other messaging systems in the sense that it is fully distributed from the ground up. The main abstraction in Kafka is a "topic", which represents a category or feed name to which messages are published. Unlike other messaging systems, a topic is further partitioned into a set of "partitions" that hold the messages of that topic. Within a partition

messages are ordered and each message is assigned a sequential ID, called the offset, that uniquely identifies each message within the partition. Typically, Kafka is run as a cluster comprised of one or more servers. Partitions are distributed among these servers, allowing a single logical topic to scale beyond a size that fits on a single server and providing a convenient level of parallelism. A Partition can also be replicated across a configurable number of servers for fault tolerance.

In Kafka, a producer is responsible for choosing which message to assign to which partition within the topic. This can be done in a round-robin fashion simply for load balancing or according to some semantic partition function, ex. based on some key in the message. Although the per-partition ordering combined with the ability to partition data by key is sufficient for a wide range of applications, total ordering over messages can be achieved with a topic that has only one partition. That topic will not benefit from any parallelism.

Kafka provides a single consumer abstraction "consumer groups" that allows consumers to expose a Kafka topic as a queue or a publish subscribe system. The Kafka cluster retains all published messages whether or not they have been consumed for a configurable period of time. Kafka's performance is effectively constant with respect to data size so retaining lots of data is not a problem.

Spark Integration

Spark Streaming provides two integration approaches with Kafka. An old receiver-based approach and a new receiver-less approach. They both have different programming models, performance characteristics, and semantics guarantees.

Spark Streaming has been extensively used with Kafka in production[68]. Apart all supported streaming sources, Spark provides a receiver-less Kafka-API which can achieve exactly-once processing semantics, as apposed to a receiver based API that can lose data under failures or can be be further configured with write ahead logs (WALs) to achieve "at-least-once" processing semantics.[55]. The new approach eliminates the need for both WALs and receivers for Kafka and makes Spark and Kafka pipelines more efficient, with stronger fault-tolerance guarantees.

2.2.2 Rabbit-MQ

Rabbit-MQ is an open source message broker that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. Official client libraries to interface with the broker are available for all major programming languages. Additionally, the RabbitMQ community has created a large number of clients and development tools covering a variety of platforms and languages.

Overview

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Instead, the producer can only send messages to an exchange. Exchanges are very simple; on one side they receive messages from producers, and on the other side push them to queues. An exchange must know exactly what to do with a message it receives, ex. should it be appended to a particular queue, should it be appended to many queues? or should it get discarded ...etc. The rules for that are defined by the exchange type. There are a few exchange types available: direct, topic, headers and fanout. Each of these route messages to queues based on different criteria. Moreover a default exchange "identified by an empty string" allows

to specify exactly a queue name to which the message will be directly routed.

RabbitMQ supports several features including task queues, message acknowledgements, durable exchanges, messages and queues, and fair dispatching to load balance work among multiple different consumers. Additional to the basic messaging operations, several RabbitMQ servers on a local network can be clustered together, forming a single logical broke. Queues can be mirrored across several machines in a cluster, ensuring that even in the event of hardware failure queued messages are safe. For servers that need to be more loosely and unreliably connected than clustering allows, RabbitMQ also offers a federation model[14].

2.2.3 Conclusion

Apache Kafka and Rabbit-MQ have completely different models and support different use-cases. Although originally envisioned for log processing, Kafka is very convenient for big data applications due to the performance and distribution guarantees that it provides. From another perspective, Rabbit-MQ speaks several protocols and have multiple features which could be switched on or off based on the user and application requirements.

2.3 Data Persistence

Big data analytics is concerned with examining the large data sets to uncover hidden patterns, unknown correlations, market trends and other useful information. Obviously there are different kinds of analysis that could be performed on different data-sets. The analysis to be performed could be known in advance before data creation or could be developed while data is being created (e.g. as a result a previous analysis, a new technology or business requirement), or even after the data has been collected. In the context of data storage, persistence implies that data survives after the process with

which it was created has ended, i.e. storage on a non-volatile media. This allows newly developed or unanticipated analysis to be performed on data that was generated beforehand.

In our system, we perform a kind of predictive analytics on the data as it arrives. For newly developed analytics and data-driven programs to make use of former data, we support data persistence on disk. Using current technologies, large scale data storage could be achieved at very high speeds, given adequate data modelling and configuration. We have chosen Apache Cassandra for data persistence among various alternatives including Apache HBase[31] and Mongo-DB[45]. Cassandra's data-model perfectly fits the time-series data upon which we base our analysis. Following, we give an overview of Apache Cassandra and we present the rational behind our choice.

2.3.1 Apache Cassandra

Cassandra is an open source, non-relational database that offers continuous availability, linear scale performance and operational simplicity[12]. It was originally developed at Facebook[23] in order to meet the applications strict operational requirements in terms of reliability and scalability. It was designed to fulfill the storage needs of the Inbox Search problem [24], which requires to handle very high write throughputs, while not sacrificing read efficiency and scale with the number of users. Moreover, a Cassandra cluster can span multiple data centers and cloud availability zones, and doesn't have a single point of failure[25]. Cassandra offers a flexible and dynamic data model with features including data protection, data compression and tunable data consistency which allow is to serve as a general purpose solution[25], beyond its initial use in social media.

Cassandra is used within our framework in order to provide in-memory and disk storage. Its main usage is to persist the streamed environments state

providing a historical data archive, while being highly available for interactive data querying and manipulation. The use of persisted data is however, external to our system. Cassandra was the preferred persistence technology mainly for the following two reasons

1. Cassandra's data model is an excellent fit for handling data in sequence regardless of data type or size. When writing data to Cassandra, data is sorted and written sequentially to disk. When retrieving data by row key and then by range, we get a fast and efficient access pattern due to minimal disk seeks. The streamed time series data is an excellent fit for this type of pattern [27].

2. Due to its very high write through-puts, Cassandra is perfect for consuming lots of fast incoming data from devices, sensors and similar mechanisms that exist in many different locations[12].

2.4 Data Visualization

Data visualization fit everywhere in the big-data pipeline, thus adds a lot of benefit for big data systems. Visualization techniques have made it somewhat easier to gain understanding of the raw and processed data, as well as the processing stages and the big data system. We have chosen to use a visualization tool called Lightning[38]. Lightning has been developed to serve a very similar system called Thunder, that we have described in 3.2.1. Thunder is designed for large-scale analysis of time-series (neural) data and is built on top of Spark, which makes Lightning a perfect fit for our data and the Spark ecosystem.

2.4.1 Lightning-viz

Lightning is a data-visualization server that provides API-based access to reproducible, web-based, interactive visualizations. It is designed to work

with large data sets and continuously updating streams. Lightning provides a core set of visualization types, including streamed visualizations and is built for extensibility and customization. We have chosen to use the Lightning server because it is very well suited for time-series data and is designed to run within the Spark ecosystem serving the Thunder project. The Lightning server is open source[40] and has official Python, Node.js and Scala clients. Within our work in the project, we have contributed in the official Lightning-scala API[41]; we describe the integration with the Lightning-server and our contribution in 5.4.4.

2.5 Conclusion

Today's leading deployments combine three distributed systems to create a real-time trinity (1) A messaging system to capture and publish feeds (2) A transformation tier to distill information, enrich data and deliver the right formats, and (3) An operational database for persistence. Together, these systems create a comprehensive, real-time data pipeline and operational analytics loop[74]. We employ a similar real-time infrastructure with an additional visualization tier in order to visualize the data and its processing. We decided to use Kafka and RabbitMQ for messaging, Spark and its high-level libraries for processing, Cassandra for persistence and the Lightning-viz server for visualization. In this section we came across each of these technologies highlighting their basic features and the reasons behind their choice.

3
Related Work

Real-time processing of large amounts of sensing data normally requires very high computing abilities and large-scale hardware infrastructures. Even with sufficient resources, it is still challenging to reliably process the generated large-scale, time-stamped datasets[62]. Since the emergence of the Internet of Things, it has been highly coupled with Cloud Computing as its underlying infrastructure and various application domains, ranging from Green-IT and energy efficiency to logistics, have already started to benefit from the combination of these concepts. At the moment, the combination of the Internet of Things and Cloud Computing is a "hot topic" in discussion and research, and various models that benefits from the integration of both technologies have been proposed in various domains, like agriculture and forestry[66], health care[75] and environmental monitoring[91].

Along our literature review we did not thoroughly research the conceptual and theoretical models behind the integration of Cloud Computing and IoT, but we were more engaged with emerging applications, their main use cases and rational. Our main focus was on cloud based systems that process and analyse time-series or sequential data in real-time, regardless IoT has been considered a part of that or not. We have considered reviewing systems that have been developed on top of Apache Spark, and that have been published to the Spark-packages index, a community index of packages for Apache Spark[13]. In this Chapter we give an overview on the research that has been done in that context. We start by an introductory section on the Spark-packages index, then we describe related and inspiratory work.

3.1 Spark Packages Index

Spark is packaged with higher level libraries providing support for SQL queries, streaming data, machine learning and graph processing. These standard libraries increase the developers productivity and can be seamlessly combined to create complex work flows. Additional to these libraries are external community modules that extends and/or customizes Sparks functionality. These modules reside in the Spark-packages index.

Spark-packages index is a community website to track the growing number of open source packages and libraries that work with Apache Spark. Spark Packages makes it easy for users to find, discuss, rate, and install packages for any version of Spark, and makes it easy for developers to contribute packages. It features integrations with various data sources, management tools, higher level domain-specific libraries, machine learning algorithms, code samples, and other Spark content. Sparks packages index is a very clear demonstration of how active the Spark community is. By the time of our literature

review, the index contained 53 packages; and by the time of our writing the number of packages has almost doubled. We have surveyed the community index and were interested in three different classes of packages. We classify these into:

1. Systems that are concerned with real-time processing and analysis of Big-Data. These are very similar to our system and were great sources of inspiration.

2. Systems that provide large-scale machine learning algorithms on top of Apache Spark. While Spark includes a high-level library for distributed machine learning, which we are already utilizing, there have been various large-scale machine learning packages implemented on top of Spark. These additional packages provide learning algorithms and scenarios that have not been considered by the Spark developers, e.g. deep-learning algorithms. These could be considered for future extension to the system to perform new kinds of analytics that are not yet supported by Spark.

3. Reference applications that considers the integration of Spark with various systems. We believe the main purpose of these applications was to demonstrate the integration of Spark with other technologies like streaming sources or visualization servers. Reviewing these applications was essential to decide upon our workflow and overall architecture.

In the following sections we describe packages in each of these categories.

3.2 Real-time processing & analysis of big-data

3.2.1 Thunder

Thunder is a library for analyzing large-scale spatial and temporal data. It was developed by 'neuroscientists' within the HMMI Janelia research campus[32] targeting the analysis of large-scale neural data to enable better understanding for the brain function.

Thunder is built on top of Apache Spark and includes utilities for loading and saving data using a variety of input formats, classes for working with distributed spatial and temporal data, and modular functions for time series analysis, image processing, factorization, and model fitting. The library implements a variety of univariate and multivariate analyses with a modular, extendable structure well-suited to interactive exploration and analysis development. The authors have demonstrated how these analyses were used to find structure in large-scale neural data[73].

Thunder is a very good example that demonstrated how Spark can be extended to be more specific and optimized to particular data representations and use-cases. Being developed by neuroscientists, it is an example of ease of extension and development for Spark. Together with Thunder, the authors have built a web-based visualization engine, called Lightning, for handling large data sets and data streams. The Lightning server was used alongside Thunder for identifying structures and patterns in the data. The authors have been experimenting both Thunder and Lightning to monitor and manipulate patterns of brain activity in zebrafish and mice[72].

3.2.2 Stratio Streaming

Stratio streaming [60] is a Complex Event Processing platform built on top of Spark Streaming. It combines the power of Spark Streaming as a continuous computing framework together with Siddhi CEP engine[86] as complex event processing engine[1]. The platform consists of (1) Stratio Engine, which comprises a Kafka Cluster, Spark Streaming and Siddhi CEP engine, (2) Stratio API, both in Scala and Java, providing a simple interface to the Stratio Engine and (3) Stratio Shell, built on top of Stratio API to enable interaction with the Streaming Engine. Moreover SQL-like Stream Query Language is provided for convenient stream manipulation.

Stratio Streaming could be viewed as an abstraction layer that simplifies the use of the combination of Spark Streaming, Apache Kafka and Sidhhi CEP engine. It provides a set of operations on top of Spark Streaming that facilitates the use of streams and queries, thus automating some of the useful common tasks and operations on streams. Stratio Streaming Benefits from Spark Streaming's ability to satisfy big data applications that require mixing both batch and stream processing in an efficient, reliable and fault tolerant manner. It also adds Complex Event Processing capabilities by integrating the Siddhi CEP engine. Stratio Streaming can be used for the creation of streams and queries on the fly, sending massive data streams, building complex windows over the data or manipulating the streams in a simple way by using an SQL-like language.

The following figure provides an overview for Stratio Streaming. The API component forwards incoming user requests to the Stratio Streaming Engine component via Kafka topics. Upon receiving a message the component will

[1]Complex event processing, is event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances. CEP as a technique helps discover complex events by analyzing and correlating other events

send a KeyedMessage to a Kafka topic for which the key will be the message operation (create, select, insert, ...etc.). The Stratio streaming engine will be listening to these topics and handles the messages accordingly to their type.

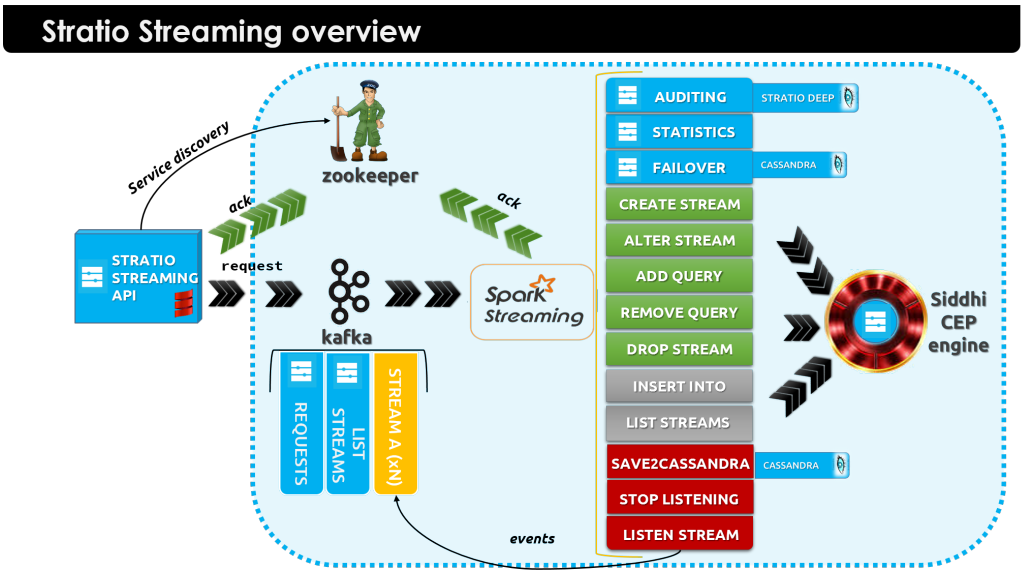

Figure 3.1: Stratio Streaming Overview.

3.3 Machine Learning Extension Packages

While Spark natively provides large-scale distributed machine learning algorithms, we have identified several packages that either extends Spark's to include supplementary algorithms and optimisations or integrates Spark with existing machine learning centered platforms. In the following sub-section we present a very interesting package that integrates Spark with H2O, an open source in-memory machine learning engine. Next we present two packages that extends Spark to include various machine-learning algorithms. While these contributions are not considered as direct related work, they can be utilized to include machine learning utilities that are not supported by Spark.

3.3.1 Sparkling Water

Sparkling Water[56] is an existing collaboration to integrate Apache Spark with H2O, an open source in-memory ML engine developed by 0xdata[1]. The integration extends Sparks machine learning with H2O algorithms, leaving the user with enough options to choose his favorite algorithm, whether it is from H2O or MLib.

H2O[30] is an open source parallel processing machine learning engine written in Java. It provides a set of state-of-the-art machine learning and deep learning algorithms that are efficiently performed in memory. H2O can run on a single machine or a cluster, either on a local network, or a cloud,. It can be accessed via a web-based user interface or one of its API's, currently in R, Java, Scala and Python. H2O can connect to data from HDFS, S3, SQL and NoSQL data sources and integrates with Excel, R studio, Tableau and more.

Sparkling Water is a very smart integration between Apache Spark and H2O. Instead of re-implementing H2O algorithms in Spark, the integration allows the creation of H2O instances within Spark's workers in order to perform the H2O algorithms allowing both engines to work collaboratively in a spark environment. A second cut was the use of Tachyon in-memory file system[78] to transfer data back and forth between Spark RDDs and H2O. To further simplify the transition an H2ORDD was added as a new RDD type in Spark. This allowed to seamlessly move data back and forth between Spark and H2O.

Running the H2O software directly in the Spark cluster required few changes in the Spark interface. These changes are related to Sparkling Water cluster formation. The approach was to embed a full H2O instance inside the Spark Executor JVM, and the H2O instances need to find each other during application initialization. This also enabled using the spark-submit approach

to pass a sparkling water application jar file directly to the Spark Master node and distributed around the Spark cluster. The following describes the sparkling application life cycle:

1. The existing spark-submit command is used to submit the Sparkling Water application jar file to the Spark Master node.

2. The Master JVM distributes the application jar to each of the Spark Worker nodes.

3. Each Spark Worker starts a Spark Executor JVM.

4. Each Executor starts an H2O instance within the Executor. This H2O instance shares the JVM heap with the Executor since it is embedded, but creates its own Fork/Join threads for CPU work. The Sparkling Water cluster fully forms once all the Spark Executor JVMs bring up their embedded H2O instances.

After completing the above steps, the application's main Scala program runs, giving the user full access to both the Spark and H2O environments in one unified program flow of control.

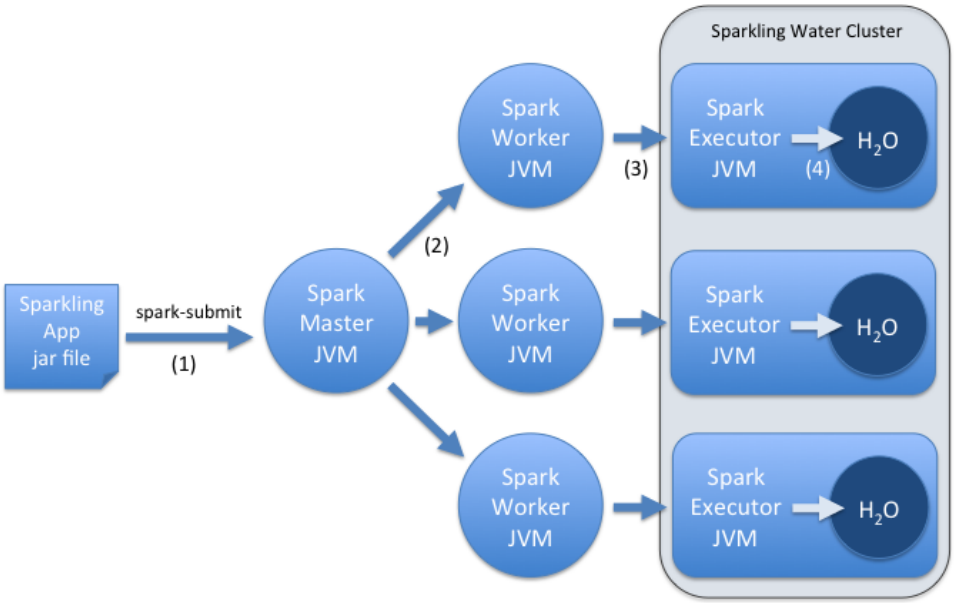

Figure 3.2: Sparkling Water Application Life Cycle

3.3.2 Aerosolve

Aerosolve[2] is a machine-learning library implemented on-top of Spark's core module. It provides sophisticated machine learning features, such as geo-based features, controllable quantization and feature interaction.

Aerosolve is a tool developed and used by Airbnb[3], a website for people to list, find, and rent lodging. Aerosolve is used to help people figure out the best price for their Airbnb rooms and apartments. This library is meant to be used with sparse, interpretable features such as those that commonly occur in search (search keywords, filters) or pricing (number of rooms, location, price) and is not as interpretable with problems with very dense non-human

interpretable features such as raw pixels or audio samples. The library is designed from the ground up to be human friendly and doesn't make use of native Spark's machine learning module.

3.3.3 Zen

Zen[65] is a large scale machine learning platform that is built on top of Spark. It includes several algorithms including logistic regression, latent dirichlet allocation (LDA), factorization machines, and deep neural networks (DNN). We include Zen in our review as an example of a machine learning package that extends Spark's machine learning module with sophisticated optimizations and newly added features. Zen also makes use of Sparks GraphX library.

3.4 Reference Applications

In this section we present three reference applications that demonstrates the integration of Spark with various technologies. These applications were great sources of inspiration and provides very similar scenarios to our considered use-cases.

3.4.1 Spark-ml-streaming

Spark-ml-streaming[54] is a Python application that generates data, analyzes it in Spark Streaming, and visualizes the results with Lightning2.4.1. The analyses use streaming machine learning algorithms included with Spark. On a very high-level this application performs a very similar task to our system, where input data-streams are processed and analysed using Spark's streaming machine learning algorithms and then visualized using the Lightning visualization server.

3.4.2 Meetup Stream

Meetup Stream[43] is a simplified application demonstrating the integration of Spark, Spark Streaming and Spark Machine Learning to provide social connections recommendations based on meetup.com[44] rsvp stream. The application is implemented in scala, and includes various utility source-code for implementing custom streaming receivers, stateful-operators, broadcast variables as well as combining batch and stream processing with machine learning. In the following figure we present the implemented recommendations pipeline.

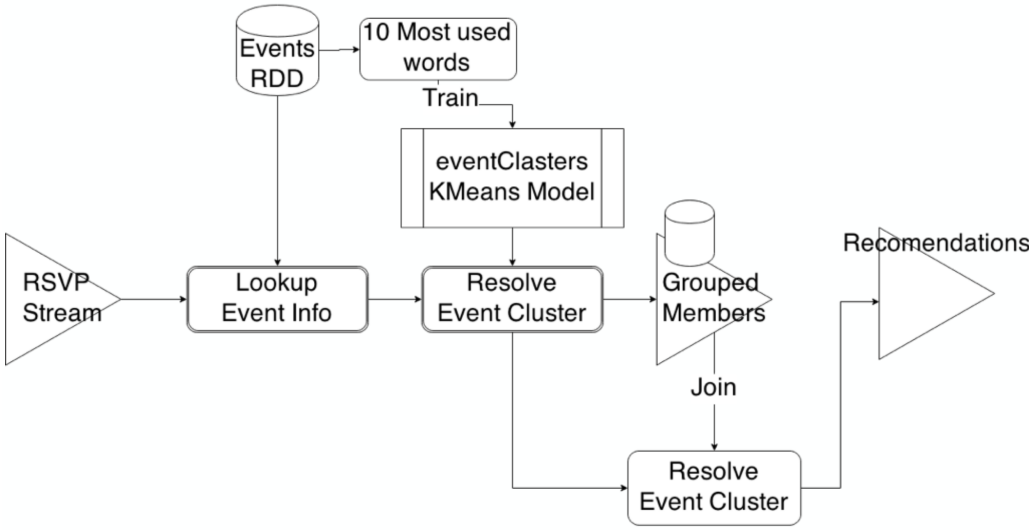

Figure 3.3: Meetup Recommendations Pipeline

3.4.3 KillrWeather

KillrWeather[36] is a reference application showing how to easily leverage and integrate Apache Spark, Apache Cassandra, and Apache Kafka for fast, streaming computations on time series data in asynchronous Akka event-driven environments. It combines fast access to historical data with streamed

real-time weather data on the fly for predictive modeling. This application serves a very similar purpose to our system, where streamed time-series data is processed and analyzed and could be then be queried later. The application is however, missing an analytics component and is confined to data aggregation tasks. The application is bundled with a data-ingestion server and a client that runs queries against the raw and the aggregated data from the ingested Kafka stream.

4

API

The basic component we provide is a software library (API) that provides a set of common manipulation functions for the domain of time-series. The API is implemented in Scala, as an independent component on top of Apache Spark and is further utilized by our system in order to analyse the received sensor data.

In Spark we model a time-series as an RDD of time-stamped values, i.e. RDD[(timestamp: Long, value: Double)], and we provide three abstractions that represent different forms a time series could have. These are TimeStampedValueRDD, SampledTimeStampedValueRDD and TimeStampedTransitionsRDD. In this chapter, we start by a brief overview on time-series and the kind of analysis we perform on the time-series data, then we describe the

API abstractions and the methods they provide.

4.1 Time-series Overview

Among all the types of big data, data from sensors is referred to as time-series[33] and is the type of data we analyze in our system. In this section we provide a very concise overview of time-series data and time-series analysis, and we introduce the type of analysis that we perform in our system, which is a form of regression analysis and shall not called "time-series analysis".

4.1.1 Time-series and Time-series Analysis

A time-series is an ordered sequence of observations of a well-defined data item obtained through repeated measurements over time. Time-series data have a natural temporal ordering. This makes time series analysis distinct from other data domains like cross-sectional and spatial data domains in which there is no natural ordering of the observations. Time series analysis accounts for the fact that data points taken over time may have an internal structure, such as auto-correlation, trend or seasonal variation, that should be accounted for. Time series analysis shall also help obtaining an understanding of the underlying forces and structure that produced the observed data. Time series models are uniquely suited to capture these characteristics.

4.1.2 Regression Analysis on Time-series

In our system, we perform a kind of regression analysis on the time-series data. We employ regression analysis to test how the current values of one or more independent times series affect the current value of another time series. In the corresponding regression model, each data point (feature vector) is an independent example of the concept to be learned, and the ordering of data points within a data set (the whole training set) does not matter. This type

of analysis on time-series is not called "time series analysis", which focus on comparing values of a single time series or multiple dependent time series at different points of time[61].

4.2 Provided Abstractions

The main purpose of the API is to provide a set of common manipulation functions for the domain of time-series. We provide three abstractions corresponding to different forms a time-series could have, with each containing common transformations that could be performed on the time-series in that form. In the API, we assume values of a time-series are recorded at regular intervals over a well defined time-period. To support this assumption, we provide a utility abstraction "SamplingInterval" that represents a sampled time-interval. The SamplingInterval is an essential component in the definition of each of our time-series abstractions. In this section we describe each our abstractions and their supported methods in details. We end this section by an implementation notice about extending Spark RDDs.

4.2.1 Sampling Interval (fromTimestamp, toTimestamp, increment)

A requisite abstraction we provide in the API is the SamplingInterval. A sampling interval represents a time interval on which a group of associated time-stamped values, e.g. an RDD[(timestamp: Long, value: Double)], shall be sampled, i.e. time-stamps of the time-stamped values shall lie on sampling points within that sampling interval. A sampling interval is used to model the frequency of data collection of the associated group of time-stamped values.

We provide three common operations for the SamplingInterval that we have

been using repetitively in our API. These are getSamplingPoints, upsample and downsample, and are described as following:

A. getSamplingPoints : List[Long]

The most desirable operation on a sampling interval that we provide is to enumerate its sampling points. The sampling points for a sampling interval are points that lies within the sampling interval, starting by the *fromTimestamp* and increasing by *increment* until reaching *toTimestamp*. In some cases, the *toTimestamp* is not a sampling point its-self, i.e. it will not be reached if we have started from the *fromTimestamp* and kept moving ahead by *increment*, in these cases, the last sampling point considered is the one reached right before *toTimestamp*.

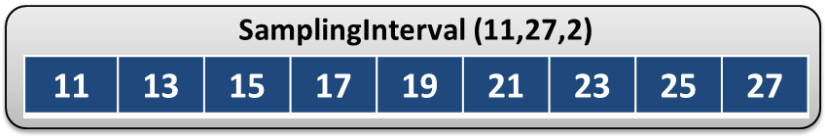

Figure 4.1: A SamplingInterval and its sampling points

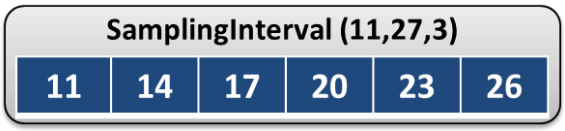

Figure 4.2: A SamplingInterval and its sampling points. Note that the *toTimestamp* is not included since it's not a sampling point its-self

B. upsample : SamplingInterval

This function returns a new up-sampled sampling interval, having approximately twice the original sampling points. Given a SamplingInterval with an *increment*=x, this functions results in a new SamplingInterval with an *increment*=x/2

C. downsample : SamplingInterval

This function returns a new down-sampled sampling interval, having approximately half the original sampling points. Given a SamplingInterval with an *increment*=x, this functions results in a new SamplingInterval with an *increment*=x*2

4.2.2 TimeStampedValueRDD (rdd, samplingInterval)

A TimeStampedValueRDD models an RDD of time-stamped values. It extends RDD[(Long, Double)] corresponding to the (time-stamp, value) respectively and has a sampling interval that it follows. A TimeStampedValueRDD follows its sampling interval in the sense that all its elements are bound within that sampling interval; each element lies on a sampling point of that sampling interval and two elements can not lie on the same sampling point. A TimeStampedValueRDD, however, does not necessarily "strictly" follow its sampling interval. A TimeStampedValueRDD "strictly" follows its sampling interval if and only if, it contains exactly one element for each sampling point contained within the sampling interval. In this case the TimeStampedValueRDD is a SampledTimeStampedValueRDD. The following figure provides a visual aid to our description. In this figure we present (A) valid TimeStampedValueRDD (B) Invalid TimeStampedValueRDD (C) valid SampledTimeStampedValueRDD. (B) is an invalid TimeStampedValueRDD since 18 (timestamp marked in red) is not a sampling point on the sampling interval (11,27,2).

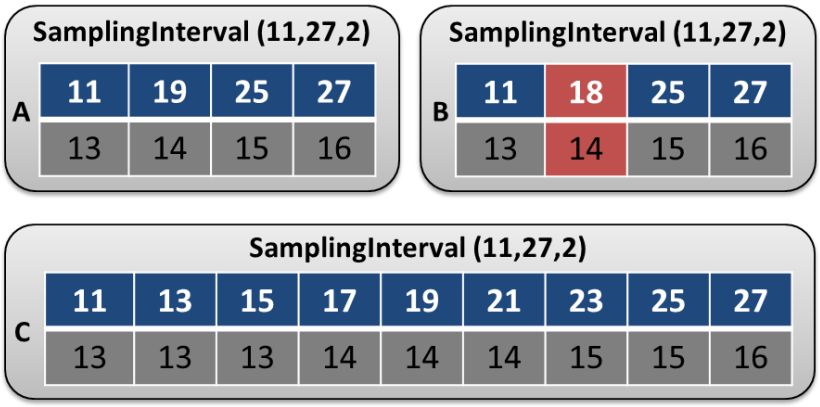

Figure 4.3: Figure demonstrating the concepts of TimeStampedValueRDD and SampledTimeStampedValueRDD

TimeStampedValueRDD is the basic time-series abstraction we provide. The common operations we provide for a TimeStampedValueRDD are re-sample, isSampled, toSampledTimeStampedValueRDD and mapValues. Additionally, for convenience, we provide an implicit conversion function, that converts an RDD[(Long, Double)] to a TimeStampedValueRDD. This shall be the only way to create a TimeStampedValueRDD from an RDD[(Long, Double)], since it guarantees that the resulting TimeStampedValueRDD follows its sampling interval. Before we describe the implemented operations, we introduce the notion of a "resamplingFunction", which is a general function format that is extensively used as an input for higher-order functions implemented for the TimeStampedValueRDD.

resamplingFunction(Long, SamplingInterval):List[Long]:- A re-sampling function takes a *timestamp* and a *samplingInterval* as inputs and re-samples the *timestamp* on the *samplingInterval*. We consider three different use-cases for the general re-sampling function.

1. **sampling:** A sampling function expects a *timestamp* value that lies within the input *samplingInterval* but doesn't necessarily on a sampling

point on that *samplingInterval*. The function maps the *timestamp* to the closest sampling point on the *samplingInterval*. In some cases it might be desirable to map the single *timestamp* value to more than one sampling point on the *samplingInterval*, e.g. if it exists mid-way between the two sampling points. Some implementations however could always map a *timestamp* value to exactly one sampling point. We support both cases by allowing the re-sampling function to return a list of values instead of a single value.

Within the API we provide three sampling functions, these are sampleLeft, sampleRight and sampleAccurate. As the names imply, sampleLeft maps the *timestamp* value to the closest smaller sampling point; sampleRight maps the *timestamp* value to the closest larger sampling point, and sampleAccurate maps the *timestamp* value to the closest sampling point(s). Note that the resulting time-stamps will always be on sampling points of the input *samplingInterval*.

2. **downsampling:** The down-sampling function expects a *timestamp* value that lies within the *samplingInterval* but doesn't necessarily lie on sampling point on that *samplingInterval*. The function samples the input *timestamp* value on a down-sampled version of the input *samplingInterval*. This function could be achieved by calling a sampling function on the *timestamp* value and the down-sampled version of the *samplingInterval*. We include a downsampling function within the API that uses sampleAccurate function described above. Note that the resulting time-stamps will always be on sampling points of the input *samplingInterval* as well as its down-sampled version.

3. **upsampling:** The up-sampling function expects a *timestamp* value that lies on a sampling point on the *samplingInterval* and creates new sampling points that lies on an up-sampled version of the input *sam-*

plingInterval from that time-stamp value. We provide three different implementations of the upsampling function within the API, these are upsampleRight, upsampleLeft and upsampleAccurate. As the names imply, upsampleRight creates a new sampling point on the right of the input *timestamp*, upsampleLeft creates a new sampling point on the left of the input *timestamp* and upsampleAccurate creates two new sampling points on the left and the right of the input *timestamp*. The three functions includes the original *timestamp* in the resulting list. Note that the resulting time-stamps will always be be sampling points on the up-sampled version of the input sampling interval.

All the examples used in this document, considers using the re-sampling functions: sampleAccurate, downsample and upsampleAccurate.

A. resample(resamplingFunction,newSamplingInterval)

This is a higher-order function that takes a re-sampling function and a new sampling interval as inputs. It applies the re-sampling function on time-stamps of each RDD element and the current sampling interval and creates a new TimeStampedValueRDD from the mapped time-stamps. Elements in the new TimeStampedValueRDD that have similar time-stamp values are grouped together by taking an average on their values, otherwise the resulting TimeStampedValueRDD will become invalid, i.e. it won't conform to our definition above. The resulting TimeStampedValueRDD shall follow the new sampling interval. Depending on the re-sampling function this high-order operation could be used very differently, ex. in downsampling or upsampling the given TimeStampedValueRDD. The figure below provides a visual example that shows the result of using the resample operator for down-sampling a TimeStampedValueRDD. In this example we assume the down-sampling function implemented in the API is the input to the resample operator.

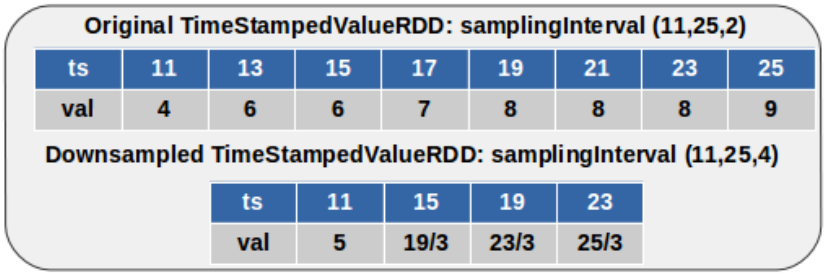

Figure 4.4: Calling resmaple(downsamplingFunction, SamplingInterval(11,25,4) on a TimeStampedValueRDD that has an original SamplingInterval(11,25,2)

B. isSampled : Boolean

This is an action on the TimeStampedValueRDD that detects weather it strictly follows its SamplingInterval or not, i.e. is a SampledTimeStampedValueRDD or not. A SampledTimeStampedValueRDD will have exactly one element corresponding to each sampling point in the sampling interval, i.e. the number of elements in the RDD is exactly the same as the number of sampling points in the SamplingInterval.

C. toSampledTimeStampedValueRDD (downsamplingFunction, upsamplingFunction)

This is a higher-order function that converts the current TimeStampedValueRDD into a SampledTimeStampedValueRDD. We implement this operation by continuously down-sampling the TimeStampedValueRDD until it becomes sampled (i.e. isSampled returns true), while maintaining all the intermediate down-sampled RDDs. Once the TimeStampedValueRDD becomes sampled, we keep up-sampling it until reaching the original sampling rate of the original TimeStampedValueRDD. During the up-sampling process we make use of the intermediate stored RDDs using their original values whenever they exist. The original TimeStampedValueRDD and the resulting SampledTimeStampedValueRDD have the same sampling interval. This

implies that each down-sample operation performed on the TimeStamped-ValueRDD will have a corresponding SampledTimeStampedValueRDD.

D. mapValues(valueMappingFunction): TimeStampedValueRDD

For convenience we provide a mapValues function that maps the values of the TimeStampedValueRDD based on an input mapping function.

E. toTimeStampedValueRDD(samplingFunction, samplingInterval)

This is an implicit conversion function, that converts from an RDD[(Long, Double)] to a TimeStampedValueRDD. The function takes a *samplingFunction* and a *samplingInterval* as inputs and creates a TimeStampedValueRDD that follows the input *samplingInterval*. To make sure the new TimeStamped-ValueRDD follows the input *samplingInterval*, we apply the resample operator (described above in A) on the RDD using the *samplingFunction* and the *samplingInterval* as inputs.

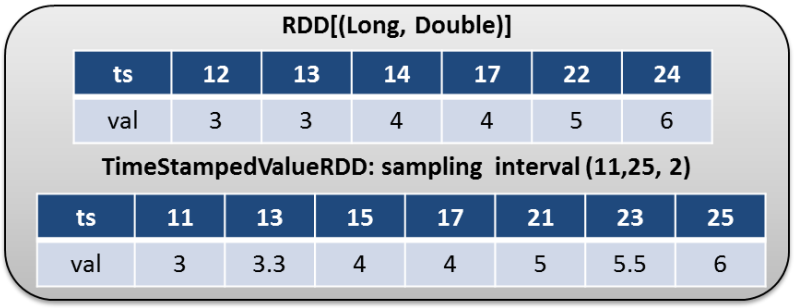

Figure 4.5: Converting the input RDD[(Long, Double)] to a TimeStamped-ValueRDD sampled along a SamplingInterval(11,25,2). We assume that the used sampling function is sampleAccurate

4.2.3 SampledTimeStampedValueRDD(rdd, samplingInterval)

A SampledTimeStampedValueRDD is a TimeStampedValueRDD that strictly follows its sampling interval, i.e. for every sampling point within the given sampling interval there exists a single time-stamped element in the RDD. No elements exist on random points within or outside the sampling interval. A SampledTimeStampedValueRDD can only be created from a TimeStampedValueRDD by calling *toSampledTimeStampedValueRDD* operation, and it maintains the sampling interval of the TimeStampedValueRDD. We provide three common operations for the SampledTimeStampedValueRDD, these are sampleByValue, findAllTransitions and findSuccessiveTransitions and are described as following:

A. sampleByValue : TimeStampedValueRDD

Converts the SampledTimeStampedValueRDD to a TimeStampedValueRDD that contains one element for each value's new occurrence, time-stamped by the timestamp of the first appearance of that values occurrence in the RDD. If a value occurs more than once at successive time-stamps only the first appearance will be identified and considered. However, if the value's next occurrence was separated by other values, its new occurrences will be considered as well. We have implemented several algorithms to achieve this function. However, its overall complexity requires at least a single shuffle and a small number of maps.

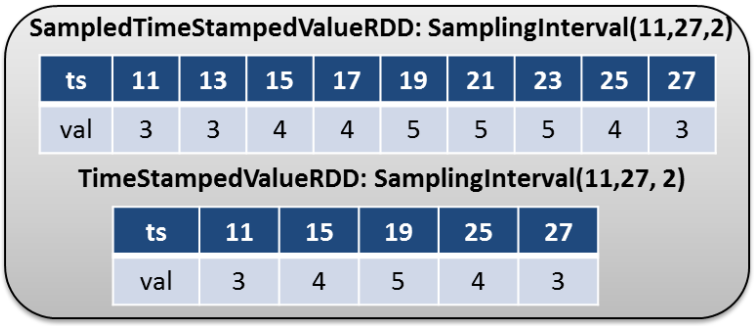

Figure 4.6: Sample by value example

B. findAllTransitions :TimeStampedTransitionsRDD

Converts the SampledTimeStampedValueRDD into a TimeStampedTransitionsRDD, containing all the transitions in the SampledTimeStampedValueRDD. A transition is simply a change in the value. Each element in the TimeStampedTransitionsRDD represents a transition from one value to another, it also includes the time-stamps of the first occurrence of each value. We achieve such primitive by calling sampleByValue on the SampledTimeStampedRDD followed by a cartesian-product on the resulting RDD with its-self. We filter the final TimeStampedTransitionsRDD to include only transitions in an increasing timestamp order.

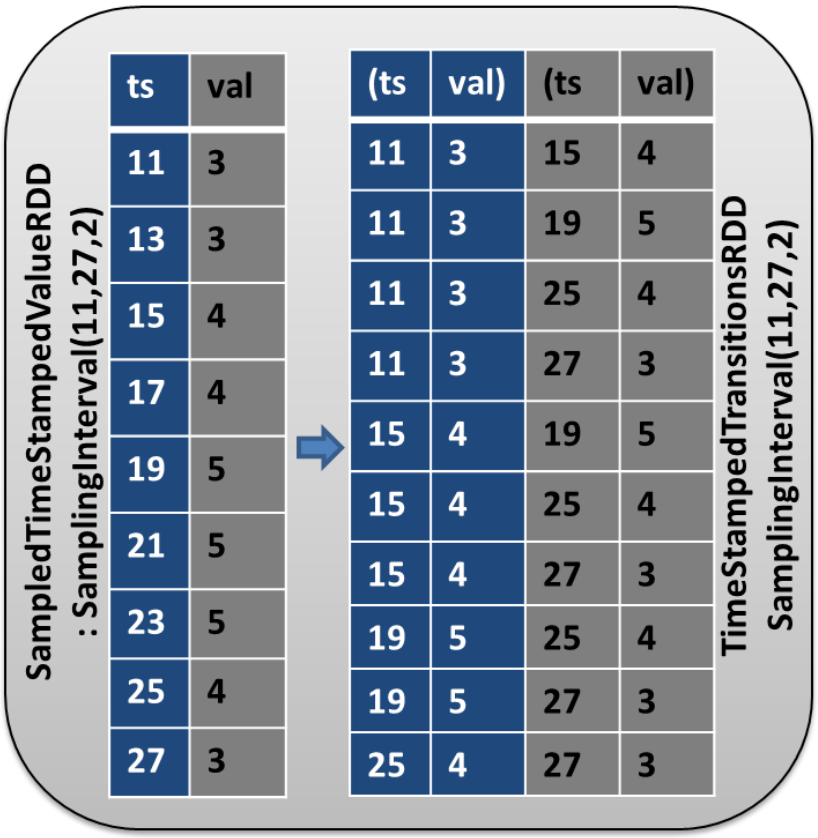

Figure 4.7: find all transitions example

C. findSuccessiveTransitions : TimeStampedTransitionsRDD

Similar to the findAllTransitions primitive, however, the resulting TimeStampedTransitionsRDD contains only successive transitions.

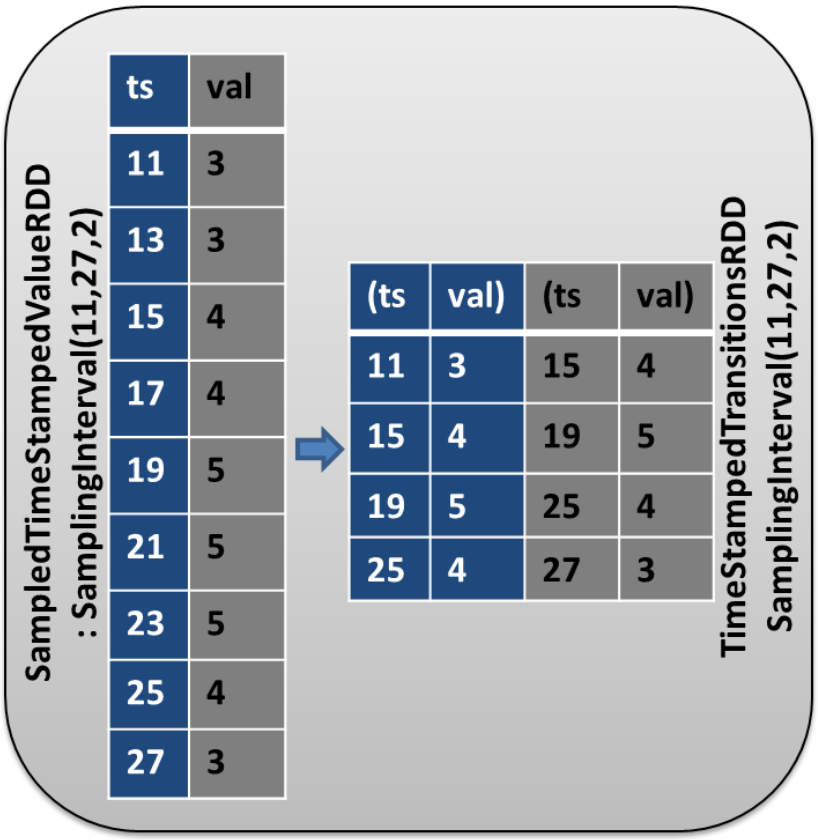

Figure 4.8: find successive transitions example

4.2.4 TimeStampedTransitionsRDD(rdd, samplingInterval)

TimeStampedTransitionsRDD is an RDD of time-stamped transitions. It extends an RDD[((Long, Double), (Long, Double))] representing (fromTimeStampedValue, toTimeStampedValue) in a transition respectively. A TimeStampedTransitionsRDD can only be created from a SampledTimeStampedValueRDD by calling *findAllTransitions* or *findSuccessiveTransitions* operation, and it maintains the sampling interval of the SampledTimeStampedValueRDD.

findTimeStampedTransitionAverage(sampledTimeStampedValueRDD): RDD[(((Long, Double), (Long, Double)), Double)]

Given the current TimeStampedTransitionsRDD, this function takes as an input a SampledTimeStampedValueRDD and returns a new RDD that contains exactly the same elements as the original TimeStampedTransitionsRDD. Each transition element however is coupled with an additional value representing the average value of the SampledtimeStampedValueRDD along the time of the transition. It's assumed that the input sampleTimeStampedValueRDD has the same sampling interval as the TimeStampedTransitionsRDD.

4.2.5 Implementation Notice: Extending the RDD

We can extend the spark API (its RDDs) in two ways: (1) by adding custom operators for an existing RDD or (2) by creating our own RDD. The former method is much simpler and is legitimate if we want to extend an existing RDD by some actions, however, in situations where we want to represent lazy evaluated transformations, we need an RDD that represents the laziness. In our API we introduce three new types of RDDs: TimeStampedValueRDD, SampledTimeStampedValueRDD and TimeStampedTransitionsRDD. Each of these RDDs extends Sparks RDDs with domain specific operators, mostly transformations, necessary to solve the problem in hand, while being general enough as common manipulation functions for our target time series domain. Moreover, we add a custom operator for the existing RDD[(Long, Double)] to convert it to a TimeStampedValueRDD. Details about how to extend an RDD can be found in the blog posts in [22] and [21].

5
SMART

SMART is a distributed application for monitoring, analyzing and predicting the behaviour of state-full real-world entities that are capable of measuring and communicating their own-state. SMART is implemented in Scala, as a higher level library on top of Apache-Spark and integrates with Apache-Kafka and Rabbit-MQ for I/O, Apache-Cassandra for data persistence, and Lightning-viz for data visualization. SMART can be configured by providing a JSON configuration file containing information about the entities to be monitored and the kind of analysis the user wishes to perform on those entities. Alternatively, a smart configuration object could be created for the same purpose.

In this chapter, we describe the system and its functionality in full details.

In the first section, we provide verbose description for the system and its features. In section 2, we describe the systems operation and workflow. Sections 3 and 4 includes relevant technical details. In section 3, the main concern is to highlight several technical aspects that we have considered in our implementation. In section 4, we describe how we did the integration of Spark with each of the complementary technologies.

5.1 System Description

SMART's core operation is to analyze large-scale time-series data describing the state of one or more user-defined target environments. The system operates on a 24/7 basis; it periodically streams-in the defined target environments state and is typically configured to learn about specific behaviours for the target environments. The system can then be used to query those behaviours in the future. During its operation, the system continuously assess and validate its learning capabilities and can only be used to predict the learned environments behaviours whenever it passes cross-validation tests defined by the user. Additionally, the system persists the streamed-in environment state for backup and future reference, and capable of producing line-charts in a streaming-fashion to visualize how different environment state-variables change with respect to each other. These are useful to perform manual exploratory analysis on the behaviour of state-variables of interest, and continuously monitor the environments state. In this section we describe the main ingredients that constitutes our system model and we present the main features of the system. At the end of this section we present the systems architecture.

5.1.1 Target Environment

The target environment represents the physical environment that shall be monitored and investigated by the system. A target environment is defined

by of a set of environment state-variables and attributes whose values make up the environments state at a given point of time. It is assumed that a target environment is equipped by several sensors that continuously measure the values of its state-variables and stream them into the system via one of its input methods. The environment attributes are constants and are directly fed into the system.

Environment Instance

While the target environment models an abstract entity, an environment instance represents an instantiation of the target environment. An environment instance is defined by specifying values for the constant target environment attributes and a set of sensors that are expected to periodically send the latest environment state-variables values to the system. Exactly one sensor for each target environment state-variable shall be defined.

Environments State

The values of the target environments attributes and state-variables at a given point of time make up the environments state at that point. While the environment attributes are constants and are defined only once with each environment instance, target environment state-variables are continuously changing. The system shall be continuously and precisely aware of those changes in order to properly and accurately learn about the environments behaviour.

Environments Behaviour

It is assumed that different environment state-variables could have some dependency relations with each others and with other attributes. This means that changes in an environment state-variable can depend on the values of other environment variables and attributes. The rate of change of an envi-

ronment state-variable with respect to other environment state-variables and attributes defines a specific behaviour of the environment.

5.1.2 Prediction Problems (Linear Regression)

Typically the system is configured to learn about specific behaviours of the defined target environment by specifying one or more prediction problems. A prediction problem can be thought of as a request for the system to build and train a "linear regression model" for a specific target environment behaviour. It is defined by specifying the environments state-variable of interest, which we call the goal-variable, and a set of non-goal state-variables and attributes, these corresponds to the dependent and explanatory variables of the linear-regression model respectively.

Learning the Rate of Change

In our description to the prediction problems, we mentioned that the goal variable corresponds to the dependable variable in the linear regression model. This is not exactly the case. In the system we are interested in rates of change rather than actual values. The main question for our prediction problems takes the following form: How much time will it take to change the goal variable from [x] to [x+1] given the current state of the environment. This also adds the restriction that for any goal variable, the system considers only unit changes and can predict unit changes. We do not limit the user queries however. If the user wants to predict the time it takes to change the goal variable from [x] to [x+n] we split that query into n queries predicting the change from [x] to [x+1], [x+1] to [x+2], ... [x+n-1] to [x+n], and respond with the total predicted time, i.e.$\sum_{i=0}^{n-1} predicted\_time(x+i, x+i+1)$. Additionally, we provide room for experimenting how different environment variables and attributes affect the goal variable, by allowing the user to explicitly change any of the environments state-variables and attributes, including the goal

variable itself. This does not change the environment state but adjusts the created queries to include the explicitly provided variables/attributes instead of those defined by the environments state.

Continuous Assessment & Cross Validation

Typically the system starts with zero-knowledge about the target environment and is expected to be inaccurate with early prediction requests. With each prediction problem we allow the system user to specify a number of batches (nB) and a maximal error rate in minutes (E) for cross-validation. Along its operation, the system continuously assess its accuracy and preserves the estimated error for the last nB batches. The system will only respond to prediction requests if and only if the estimated error of each of the last nB batches is lower than the specified maximal error rate E.

5.1.3 Latency

Along its operation, the system considers two types of latencies that shall be defined by the user. These are: streaming latency and learning latency. The streaming latency defines the overall latency of the system. It specifies the batch interval on which Spark Streaming operates, and shall be set based on the application requirements and the available cluster resources. The learning latency defines how often the system shall update its prediction models, i.e. how often shall it transform the streamed-in sensor data into feature vectors to train the maintained linear regression models. The learning latency must be multiple of the streaming latency, and shall be large enough for sufficient number of feature vectors to be created. The learning latency shall not be too large, to allow the models to be updated with an adequate frequency.

5.1.4 System Features

Support for Multiple Environment Instances

The user can define a single target environment, however, there is not limit on the number of target environment instances that can be instantiated. The only restriction is that each environment instance must have a corresponding occurrence in real-world, occupied with exactly one sensor for each defined state-variable. The system is built to learn about the behaviour of all the defined instances concurrently. Data from different environment instances is processed separately, however, the learning process for a specific target environment behaviour, i.e. prediction problem, aggregates data from different environment instances. The advantage of this approach is that more environment instances will bring more data to the system, thus the learning process will be much faster. This might be on the expense of accuracy in some cases. If the user did not consider environment attributes that affects the rate of change of the goal-variables, they will not be considered by the model in learning and predictions could not be very accurate. In case of learning on a single environment instance any constant values does not count, since in prediction they will still remain the same.

Data Source Integration

The system supports Apache Kafka and Rabbit-MQ as intermediate messaging queues for data input and output. A streaming source can be configured to stream-in the environments state or the user prediction requests, while a streaming sink can be configured to stream-out the prediction responses.

Data Persistence

There are numerous types of analysis that could be performed on the streamed time-series data that are not possible to cover by a single system. The system can however be configured to persist the streamed-in sensor data on a Cas-

sandra database to keep room for performing various kinds of analytics on the persisted data. Querying Cassandra, however, is external to the system.

Data Visualization

In many cases data visualization is desirable to gain insight about big data. We use Lightning-viz, a data visualization server, to provide streamed line charts visualizations for the environments state in real-time. For a given target environment, the system user can visualize how different environment state-variables change with respect to each other in real time. The user can configure the system to display subsets of the environment state-variables together or alone. This form of visualization is useful to visually examine regular time series data. Additionally, the system can be configured to display the learning accuracy i.e. error, for given prediction problems, estimated with each processed sensor data batch, and even display the learning curves of different prediction problems together. Such error display can be very helpful providing an intuition about the convergence and properties of the prediction problems.

5.1.5 System Architecture

The figure below depicts a high-level system architecture.

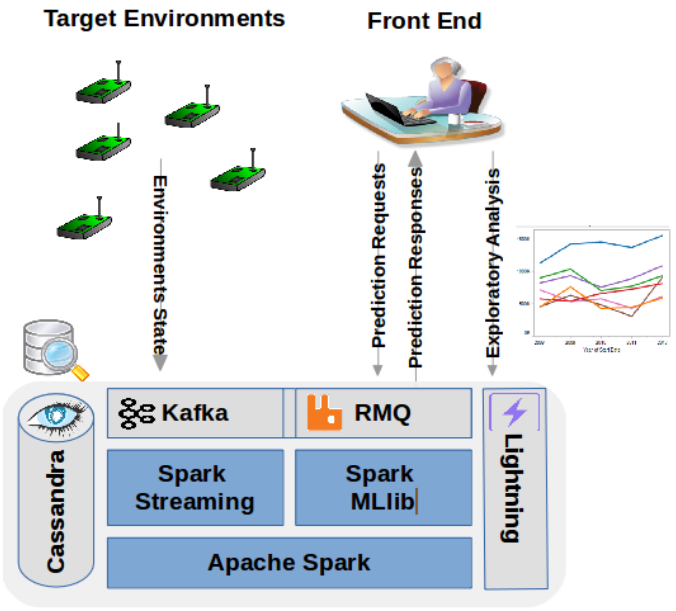

Figure 5.1: System Architecture

5.2 Application Work-flow

In this section we present the applications operation and workflow. We start by describing all the data streams maintained by the system, then we present the application work flow, highlighting the usage of each stream within the system and how the streams are created and maintained. We end this section by zooming into the process of creating feature vectors from the input sensor data stream.

5.2.1 Streams Definitions:

Along its operation the system maintains 6 different types of streams. In this subsection we describe each stream and its usage within the system. For each input/output stream to/from the system, we present a uniform JSON format that is accepted/created by the system.

Sensor Data Stream:-

This is an input stream to the system containing time-stamped sensor recordings of target environment instances states. It is received with the user defined latency *Latency.streaming* in minutes and is considered the basic stream from which all generated streams in the system are created. In a typical run the system persists the sensor data stream as it is on Cassandra for back-up and future data analytics. Here we remind that each sensor is expected to be located in a target environment instance and is responsible for measuring a single state-variable within that instance. All the sensors responsible for measuring state-variables in a target environment instance are defined by the user when defining that instance. All sensors are expected to send their data in the following JSON format:

{
"timestamp": UNIX TIMESTAMP,
"processId": UUID,
"sensorId": Unique UUID,
"instanceId": UUID,
"value": BASE64 ENCODED VALUE
}

Prediction Requests Stream:-

This is an input stream typically created by the application user, containing user queries to specific target environment instances. It has the user-defined latency *Latency.streaming* in minutes. Each prediction request shall contain a unique query id, the name of a user-defined prediction problem to query, the id of the target environment to query about and a target value for the goal variable. Typically, when a user issues a prediction request, the system creates a feature vector from the current state of the queried environment instance. However, the user could include a state override map in his prediction

request that maps from an environment state-variable name to a corresponding value to include (override) in the feature vector. In all cases the system considers predicting the time required to change the goal variable from its current/overridden value to its current/overridden value + 1 or the target goal variable if specified. Each prediction request shall have the following JSON format:

{
"queryId": unique INTEGER,
"problemName": STRING,
"teId": INTEGER,
"stateOverrid": MAP[STRING, DOUBLE],
"target" : +ve INTEGER or -1 (unspecified) }

Environments State Stream:-

This is a stream of key-value pairs generated from transforming the sensor data stream. Its purpose is to maintain an updated state for each target environment instance. The current state for each target environment instance could be visualized to do a form of explanatory analysis on the dependencies between different instance state-variables and is also used to build feature vectors to satisfy the user query requests. The environments state stream is maintained and continuously updated by applying the stateful transformation *updateStateByKey* on the sensor data stream. Each element in the environments state stream corresponds to the maintained state of a target environment instance, i.e. the number of elements in the stream is exactly the same as the number of the defined target environment instances. In each element the key is the instance id and the value is a map from each state-variable name to its latest received value. The environments state stream has the same latency as the sensor data stream.

Feature Vectors Stream:-

This is a stream of key-value pairs generated from transforming the sensor data stream. It transforms each received batch of sensor data into labeled feature vectors for training and assessing the user defined prediction problems. The feature vectors stream has the user-defined latency *Latency.learning* which is typically multiples of the sensor data stream latency *Latency.streaming*. Using Sparks stateful window operation the transformation from the sensor data stream to the feature vectors stream was possible despite the differences in latencies. Each element in the stream has a key corresponding to a prediction problem name and a value of a labeled feature vector. In 5.2.3 we describe exactly how the feature vectors stream is created.

Problems State Stream:-

This is a stream of key-value pairs generated by trying to predict the labeled feature vectors stream elements on their corresponding prediction problems regression models. Along the systems operation, it builds and maintains a generalized linear regression model for each prediction problem. With the creation of new labeled feature vectors (in batches from each arriving sensor data batch), the maintained models are first assessed by trying to predict the corresponding label for each feature vector, then are updated by training them on these labeled feature vectors. The problem state stream contains one element for each prediction problem. In each element, the key is the prediction problem name and the value is the error resulting from the predictions on the last batch of labeled feature vectors for that specific problem. The problems state stream is further used to visualize the latest accuracy of each prediction problem.

Prediction Responses Stream:-

This is an output stream created in response to the prediction requests stream. For each prediction request, a feature vector is created and the corresponding prediction-problem is queried. The query results in a prediction response. The prediction responses stream is streamed out of the system with latency *Latency.streaming*, and each element has the following JSON format:

{
"queryId": INTEGER,
"response": STRING
}

The queryId is exactly the same as the queryId in the corresponding prediction request and the response shall contain either the predicted label, "predicted time in minutes", or an error message in case the queried learning model does not pass its cross validation settings.

In the following table we summarize the system streams and their operations

Stream	Type	Usage
SensorDataStream	Input	- Persisted - Creating all generated streams
PredictionRequestStream	Input	- Query requests - Creating prediction responses stream
EnvironmentsStateStream	Generated	- Creating feature vectors for predictions - Exploratory analysis on environments states
FeatureVectorsStream	Generated	- Training and assessing prediction models
ProblemsStateStream	Generated	- Visualizing current prediction problems accuracy
PredictionsResponeStream	Output	- Answering the users queries

5.2.2 Application Workflow

The system can be initialized by accepting a smart configuration object or a JSON configuration file as an input. These shall include all the user-defined application settings as well as the application name and the Spark master URL. If non of these is provided, default configuration will be used. This sets the system to a default JSON configuration file included in the application resources. The system parses the input/default configuration object and creates immutable user configurations describing the application settings. A streaming context is then created. In case the Spark driver program is restarting from a failure, it restores its previous state from a defined checkpoint directory. The checkpoint directory is defined while creating the streaming context for the first time. We elaborate more on the use of checkpointing and fault tolerance in 5.3.1 and 5.3.2

The first thing considered by the system is broadcasting a subset of the immutable user configurations to the Spark worker nodes. These configuration objects are repetitively used in various transformations occurring at the worker nodes. By sending these objects as broadcast variables, we prevent Spark from re-sending them each time they are accessed within a transformation. This is an optimization step that is described in details in 5.3.4. The system then starts to accept the raw sensor data and the prediction requests streams. Before starting the actual processing, all the received raw sensor data is forwarded (as-is) to Cassandra for back up and future reference.

The system starts its processing by maintaining the current state for each defined target environment instance. Initially, the state is undefined and is built up from the received sensors data. A single recording from each sensor in an environment instance is sufficient to build up the state for that instance. Instances states are continuously updated with new arriving data. The environments state stream is further processed to display user-defined

state visualizations on the lightning server. For each environment instance, a separate lightning session is created and contains state visualizations for that instance.

The next step is to create feature vectors to train the defined prediction problems. At this point, it is important to mention that while the system is parsing the input/default configuration, it initializes a streaming linear regression model for each prediction problem defined by the user. The model weights can be initialized to input weights specified by the user or zeros, in case they are not specified. The defined explanatory variables for each prediction problem are mapped to constant locations in the feature vectors. This is essential, so that each time a feature vector is created for a given prediction problem, features values are mapped to the same location in the feature vector. These locations are also maintained between system runs, given that the definitions of the explanatory variables for a prediction problem are kept unchanged, i.e. their order. In case the system is recovering from an error, restoring the streaming context form the checkpoint directory shall restore the latest values for the model weights.

While the raw sensor data stream is used to create feature vectors for the defined prediction problems, it is not processed as soon as it arrives; while configuring the system, the user specifies a separate learning latency that defines how long to preserve the received raw sensor data before it is processed to feature vectors. The learning latency shall be set very carefully to allow for a sufficient number of feature vectors to be created. The preserved raw data stream is processed collectively as a batch to create the target feature vectors. Although the data for each target environment instance is processed separately to create feature vectors related to the specific target environment, feature vectors created from different environment instances are learned by the same prediction model. Before using the labeled feature vectors in train-

ing, the system tries to first use them in prediction in order to determine its current performance and accuracy for each prediction problem. Moreover a lightning session is created to visualize the problems learning curves (average error in minutes) in real time. In 5.2.3 we describe in details how feature vectors are extracted from arriving batches of raw sensor data.

The final part in the system is concerning the user prediction requests. The system accepts and processes prediction requests with the same latency as the raw-sensor data stream. This is essential to be able to (accurately) use the current state of the queried environment instance, created from the raw sensor data, in prediction. The user can post a feature vector to predict its label or ask to create the feature vector from the current state of the environment instance or a combination of both. The last case is typically the case, where a feature vector is generated from the current environment state but some of the features are overridden explicitly by the user. This gives the user full control to experiment how different environment state-variables and attributes affect the goal variable, without being restrained by their current values. The system process each prediction request by predicting the label for each corresponding feature vector and sends the prediction response back to the user in a JSON format. If the prediction problem does not pass the cross validation settings defined by the user, the system returns an error (a very large negative number) instead of the prediction.

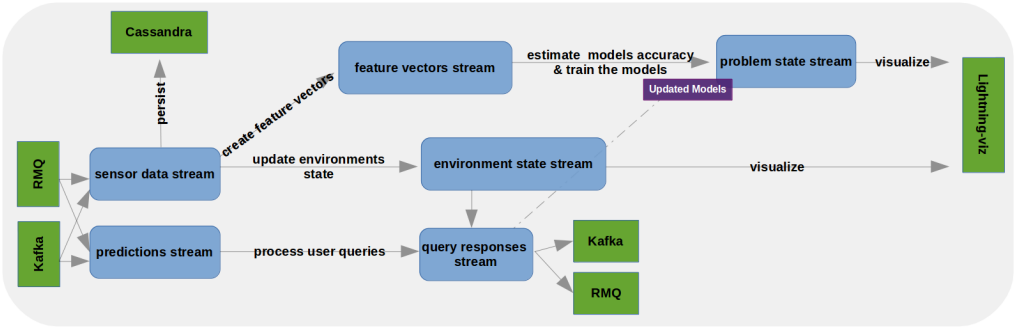

Figure 5.2: Application Workflow

5.2.3 Raw Sensor Data to Labeled Feature Vectors

In this subsection we present the set of transformations that we apply on the input sensor data stream in order to transform it to the labeled feature vectors stream. As mentioned in 5.2.1, the sensor data stream is received every Latency.streaming, but is processed every Latency.learning to create the feature vectors. We achieve this using Sparks stateful window operation. In Spark Streaming, windowed computations allow to apply transformations over a sliding window of data. In our case, we use the window operation to preserve the sensors data until the Latency.learning time passes, then we start processing. All the sensor data batches received over the previous Latency.learning are combined and processed together as a single batch.

In Spark Streaming, streamed data is processed in batches, and in the absence of stateful transformations each batch is processed independently from the previous batch. Given our windowed sensor data stream, only stateless transformations are used to convert it into a feature vectors stream. i.e. each windowed sensor data batch is transformed independently to a feature vectors batch, thus it is more convenient to continue our description in terms of RDDs instead of Streaming notation. In the following description, rely on concepts and functions defined in chapter4. Given a windowed sensor data

RDD we perform the following transformations to obtain a features vectors RDD:

1. First we split the sensor data RDD into a Map of RDDs. Each element in the map has a key: sensor id and value: RDD containing the data recordings received only from that sensor. In this step each RDD of sensor data is also transformed to a TimeStampedValueRDD. All the TimeStampedValueRDDs are creates using the same SamplingInterval as an input. The SamplingInterval is initialized by the minimum and maximum time-stamp values of the whole sensor data batch and a user-defined sampling number. The user-defined sampling number shall be specified while defining the target environment.

2. The next step is to re-sample the TimeStampedValueRDDs to fit their SamplingIntervals. This is done by converting each sensor data TimeStampedValueRDD into a SampledTimeStampedValueRDD

3. Next we create a new Map for the goal variables data. This is done by filtering the SampledTimeStampedValueRDDs Map containing data for all the environment state-variables to include only goal variables. Each goal variable SampledTimeStampedRDD is then converted into a TimeStampedTransitionsRDD by applying findSuccessiveTransitions transformation.

4. We then group each goal variable TimeStampedTransitionsRDD with SampledTimeStampedRDDs of all variables it depends on, i.e. defined as explanatory variables in a prediction problem for that goal-variable. The result of this step is an RDD for each goal variable, each element containing a goal variable transition and an iterable containing the average values for all the corresponding goal variable explanatory variables along that transition. We combine all these RDDs into a single RDD for use in the following transformations. Each element

in the combined RDD has the following form: (goalVariableSensorID: Sring, goalVariableTransition: ((timestampFrom: Long, valueFrom: Double),(timestampTo: Long, valueTo: Double)), explanatoryVariablesTransitionAverages: Iterable[(nonGoalVariableSensorId: String, nonGoalVariableTransitionAverage: Double)]). Its important to mention that each element in the RDD belongs to exactly one target environment instance.

5. In each prediction problem the user defines a goal variable and a set of explanatory variables on which the goal variable depends on. If a goal variable is defined in multiple prediction problems, it might have different explanatory variables in each. Resulting from our transformations above, each RDD element in the combined RDD contains the explanatory variables for a goal variable from all the prediction problems, thus a single RDD element shall be used to create a feature vector for each prediction problem that defines that goal variable. We apply a flat map on each combined RDD element for that purpose and the result is a key value RDD, where the key is a problem name and the value is a labeled feature vector. The feature vectors are further processed to match specific settings for each prediction problem, e.g. transformed to a polynomial feature vector, transformed to a normalized feature vectors ...etc.

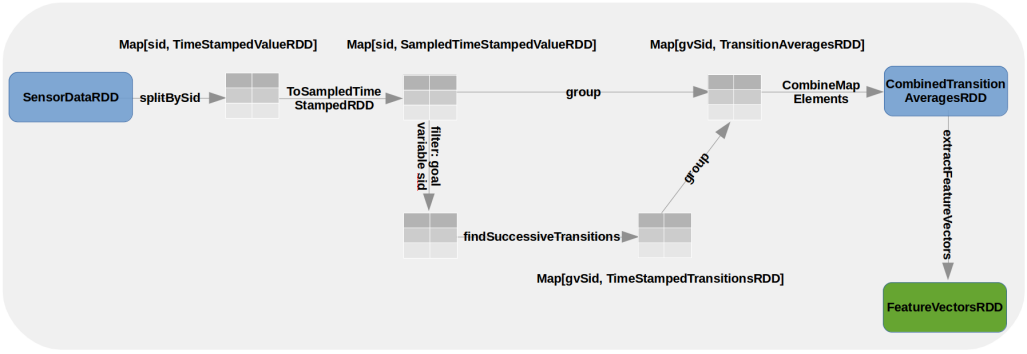

Figure 5.3: Sensor Data to Feature Vectors

5.3 Technical Details

In this section we highlight several implementation aspects that we have considered in the system. At the end of the section we provide a table summarizing the most important objects and their functionality.

5.3.1 Data Checkpointing: Preserving the Environments State

One of the features provided by the system, is to preserve the latest state for the defined target environment instances. This is essential in order to be able to visualize the state-variables for each instance in real time. Moreover, whenever a prediction request is received for one of the defined target environment instances, the latest state for that instance is used to create a corresponding input feature vector for the prediction model. We maintain the state of the defined target environment instances in a DStream of key-value pairs, where the key is the instance id and the value is a map from each instance state-variable to its latest received value. We use the updateStateByKey operation provided in Spark Streaming in order to continuously update the instances state with the latest received state data. UpdateState-

ByKey is one of the stateful transformations provided in Spark Streaming. Stateful transformations are operations on DStreams that combine DStreams RDDs across multiple batches. In our case, we combine the preserved state from the previous batch with the newly arriving state recordings in order to end-up with the updated state for each environment instance.

In stateful transformations, it is typically the case that generated RDDs depend on RDDs of previous batches, which causes the length of dependency chain to keep increasing with time (since previous batches in turn depends on the preceding ones ...etc.). In case of a failure, there could be an unbounded recovery time, proportional to the dependency chain. To over come such issue Spark provides checkpointing, where (intermediate) RDDs of stateful transformations are periodically check-pointed to reliable storage, e.g. HDFS, to cut off such dependency chains. In other words, while Spark Streaming still recomputes state using the lineage graph of transformations (in case of a failure), checkpointing controls how many previous transformations it has to consider.

In the following figure, we illustrate how we maintain the state for each target environment instance as well as the importance of check-pointing. As displayed in the figure, UpdatedState[i] is computed by combining UpdatedState[i-1] and the received SensorData[i]. In the absence of checkpointing, to recover UpdatedState[3] all the SensorData and UpdatedStates highlighted in blue are required. Considering recovering UpdatedState[100] this lineage graph will be too large and almost impossible to efficiently recover in a typical streaming application.

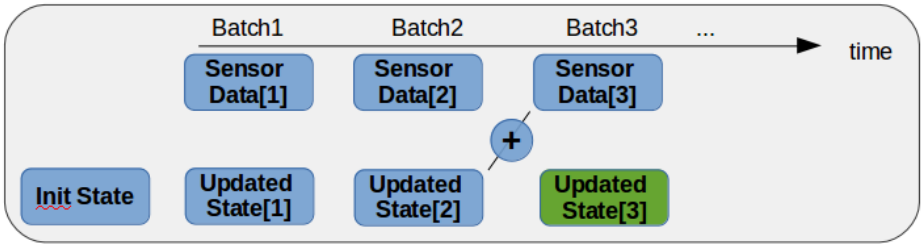

Figure 5.4: Figure illustrating checkpointing concept

5.3.2 Fault Tolerance and Preserving the System State

Checkpointing is not only useful in limiting the state that must be recomputed on failure, but also it provides fault tolerance for the driver application. If the driver program in a streaming application crashes, it can be launched again and informed to recover from a previous checkpoint. In that case Spark Streaming will read how far the previous run of the program got in processing the data and take over from there. We provide checkpointing support for DStreams that are computed using stateful operations, however, for the driver fault tolerance we provide a different mechanism that we believe is more efficient for the purpose of our application.

In our system, the only state we are interested to preserve across driver failures or subsequent system runs is the weight vectors of the linear regression models maintained by the system. Typically, weights of a linear regression model are initialized to zeros, or to weights specified by the user while defining its corresponding prediction problem. These weights are continuously updated as new sensor data is received, causing the prediction accuracy of the regression models to improve as more data is streamed into the system. The linear regression models are maintained at the driver application, and are lost in case of a driver failure or by terminating the application. In this case, subsequent system runs will not benefit from the previously streamed in data.

We provide a very simple mechanism for maintaining the state information of the linear regression models across driver failures and system runs. We set a thread at the driver application that periodically writes the current state of each linear regression model to Cassandra. The state of each regression model includes, its current weights vector, intercept and error measure. In succeeding system runs, the weight vectors could be set to initialize the weights of the prediction problems.

We take advantage of Cassandras data-modeling and we store the data in a format similar to the one used for preserving the raw-sensor data in 5.4.3. We create a wide row of data for each prediction problem. We use the prediction problem name as the row key. The number of feature vectors used in training so far as the column name (this is a monotonically increasing value, corresponding to the time attribute when storing time-series data on Cassandra), and a state string comprising the models weights vector, intercept and error measure as the column value.

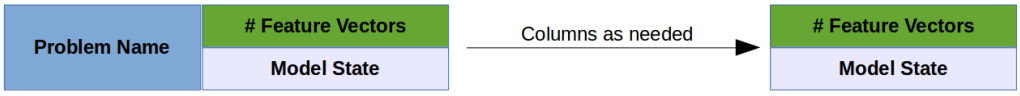

Figure 5.5: Cassandras data model for persisting the system state

5.3.3 Data Serialization: Kryo Serialization

In a Spark Streaming application, data serialization is a very important performance factor. Unlike regular Spark-Core applications in which RDDs are persisted by default as-is in memory, RDDs generated by streaming computations are persisted in memory in a seriablizeable format. Moreover, input data received by receivers, ex. from a messaging queue, are also stored in a serializeable format. This serialization obviously has overheads, e.g. the receiver must de-serialize the received data and re-serialize it using Spark's serialization format.

By default Spark serializes objects using Java Serialization. This is very flexible since it can work with any class that implements java.io.Serializable but has performance limitations. Spark can be configured to use the Kryo[37] library to serialize objects. Kryo is significantly faster and more compact than Java serialization, but does not support all serializeable types and requires registering classes used in the application in advance for best performance. We set our application to use Kryo serialization and we have registered custom implemented classes that are expected to be communicated. This configures the serializer used not only to shuffle data between worker nodes, but also when serializing RDDs to disk. This is expected to achieve a lot of performance improvements, reducing both CPU and memory overheads, also considering that we enable checkpointing which periodically stores data on disk.

In the following table we present the explicitly registered objects for the Kryo serializer and their usages within the system.

Serialized Object	Usage
SamplingInterval	Object encapsulated with each of the custom provided RDD abstractions (TimeStampedValueRDD, SampledTimeStampedValueRDD, TimeStampedTransitionsRDD) to define a sampling interval that elements in these RDDs shall follow.
PredictionProblem LearningData	Object that preserves the latest state data for each prediction problem. This includes the corresponding linear regression model, the number of feature vectors used in prediction ...etc.
Lightning	Object used to maintain session information for each Lightning session.
Visualization	Object used to maintain data about each created Lightning visualization. This object is also used to access and update streaming visualizations.

5.3.4 Broadcast Data

When a function is passed to one of Sparks distributed operations e.g. map or reduce, it is executed on remote worker nodes. Each worker node will

be working on separate copies of all the variables used in that operation. Spark automatically sends all variables referenced in a distributed operation to the worker nodes. While this is convenient, it can also be inefficient if for example the same variable is used in multiple different parallel operations, as Spark will send it separately for each.

In our system, most of the provided features requires a set of transformations that takes into account user inputs that do not change along the execution. These inputs are maintained as immutable objects that are passed as parameters to different transformations. To overcome the issue of sending the constant input objects repeatedly to the worker nodes with each transformation, we define them as broadcast variables. A broadcast variable is a read only variable that is sent exactly once to each worker node. It is cached on each worker machine and used directly rather than shipping a copy of it with each distributed operation that would access it. Spark also distributes broadcast variables using an efficient BitTorrent-like algorithm to reduce communication cost. The distribution of the broadcast variables also benefits from using the kryo serializer.

5.4 Technology Integration

In this section we present how we did the integration between Spark and our complementary technologies. We use Kafka and Rabbit-MQ for I/O, Cassandra for persistence and Lightning for visualization. Our approach was to use native spark integration if possible. If not, we look for reliable, external software packages that provides such integration for us. Our search was however confined to the Spark-packages index. At the end of this section we provide version information for each of our utilized technologies.

5.4.1 Kafka Integration

Spark Streaming provides native support to stream data from Kafka. There are two approaches for this, an old approach using receivers and Kafka's high-level API, and a new experimental approach without using receivers. They have different programming models, performance characteristics, and semantics guarantees[55].

Spark has initially provided an integration approach that utilizes Kafka's high-level API providing receivers that store Kafkas received data on Spark's executors. Jobs launched by Spark Streaming can then process the data. Under the default configuration, this approach can lose data under failures, however, to ensure zero-data loss, additional configuration is necessary. Extension package(s) exists that provides such supplement. Examples of these packages are kafka-spark-consumer[35]. By the introduction of Spark 1.3.0, a new experimental approach has been introduced that is more efficient than the receiver based approach. This approach provides simplified parallelism, exactly-one-semantics and ensures stronger end-to-end guarantees by periodically querying Kafka for the latest offsets in each topic/partition instead of using receivers to receive data.

Streaming Data from Kafka

In order to stream data from Kafka, Spark's experimental 'receiver-less' approach is used. The main disadvantage of this approach is that it does not update offsets in Zookeeper, hence Zookeeper-based Kafka monitoring tools will not show progress. However, it is possible to access the offsets processed by this approach in each batch and update Zookeeper pro-grammatically. This overcomes the limitation [55].

Streaming Data to Kafka

Although Spark provides integration approaches to stream data from Kafka, it does not provide the opposite. In order to make use of existing Spark packages, we integrated into our system spark-kafka package[53] from the Spark-packages index. This package is designed to facilitate loading batches of data from Kafka into Spark and vice verse. It contains necessary logic to support our use case but on the RDD level. The package provides a KafkaRDD abstraction that contains a method to efficiently write an RDD to Kafka. We extend such method to conform with DStreams, by saving the underlying RDDs of the DStream. The downside of this approach is that for every invocation of the function a new Kafka connection object (Kafka Producer in this case) is created for every RDD partition, which is not the most efficient way to follow. The blog post in [82] highlights further optimization to minimize the creation of connection objects when writing a stream to an external system.

5.4.2 Rabbit-MQ Integration

Spark Streaming does not provide native support to stream data to/from Rabbit-MQ in particular. At the time of our technology review, a single package that provides custom receivers for Rabbit-MQ was found in the spark packages index[49]. The package was provided by Stratio[57], a big data start-up, that is very active within the spark community. The package provides two ways for receiving data from Rabbit-MQ, (1) consuming data directly from a Rabbit-MQ queue or (2) consuming data from a Rabbit-MQ queue through a direct exchange.

Streaming Data from Rabbit-MQ

To stream data from Rabbit-MQ we relied on the Stratio Rabbit-MQ receiver package. However, we had to implement an additional extension method to

support consuming data through a Rabbit-MQ topic exchange. The provided custom receivers allow to consume data from Rabbit-MQ queue and direct exchange, but did not support consuming data from topic exchanges. Topic exchanges allow a rich class of use cases for using the messaging queue. We implemented such method by creating a queue on the fly and binding it to the topic exchange of interest on given routing keys (the topic exchange name and routing keys are inputs to the method). We then call the provided Stratio queue receiver method on the queue name.

Streaming Data to RabbitMQ

In order to stream data to Rabbit-MQ we provide our own implementation. Since we can stream-in data from Rabbit-MQ queue, direct-exchange and topic-exchange, we provide similar stream-out operations for convenience. Similar to the implementation provided for Kafka[53], we create a connection object for every RDD partition. Further optimization could be achieved by reusing connection objects across multiple RDDs/batches.

5.4.3 Cassandra Integration

The required integration with Cassandra was limited to the storage of ingested environments state data from different streaming sources. How the data shall be used afterwards is beyond the scope of this work. Spark doesn't provide native support for interaction with Cassandra. However, several packages exist in the Spark-package index that provide such integration. We identified deep-spark[58] by Stratio and Datastax Spark-cassandra connector[52] as two suitable alternatives for integration with Cassandra, and we chose the latter. The Spark-cassandra connector exposes Cassandra tables as Spark RDDs and allows saving RDDs back to Cassandra. Moreover, it integrates with Spark Streaming, providing a very convenient way to store ingested data from different streaming sources directly to Cassandra. This

perfectly fits our use case.

Persistence Data Model

Time series is one of the most compelling data models for Cassandra. It's a natural fit for the big table data model and scales well under a variety of variations. The simplest model for storing time series data is creating a wide row of data for each sensor[27]. In this case the sensor uuid is used as the row key, the timestamp of the reading will be the column name and the reading its self will be the column value. Since each column is dynamic, a row can grow as needed to accommodate the data. Cassandra can store up to 2 billion columns per row, which, given an application with latency requirements of minutes, is enough to store data for more than 3800 years, assuming a sensor sends its readings every one minute. We also get the built-in sorting of Cassandra to keep everything in order.

5.4.4 Lightning Integration

The lightning server is API driven and data can be pushed to the server via REST API. Moreover, official Python, Node.js, and Scala clients are provided. Our usage to the lightning server was confined to the generation of exploratory graphs. We generate streamed line-charts for the target environments states as well the accuracy of the maintained prediction models by the system. Since our development environment was completely based on Scala, we preferred to use the lightning-scala client for posting data to the lightning server.

The provided lightning-scala client was very naive compared to the Python and Node.js clients. The main issue we encountered is that it was missing the stream visualization functions, which all our work was based on. Along our work, we implemented the line-streaming and corresponding functions

for the scala client. We have been inspired by their implementations in the alternative Python and Node.js clients. We have sent a pull request with our addition to the official client repository on GitHub[46] and it has been merged to the repository.

5.4.5 Technology Version Information

In this sub-section we provide version information for each of our utilized technologies. Along our work, we have encountered several issues concerning visioning conflicts with our imported packages, and we believe the main reason for that is the active development in Spark, which in-turn triggers updates in the whole Spark echo system to support its new features and models. Along the time of this project, Spark has introduced 2 major releases. We have managed to keep up with the latest Spark version to make use of the introduced optimizations and features. The following table provides version information for each of our utilized technologies.

Technology	Version
Scala	2.11.7
Apache Spark	1.4.1
Apache Kafka	0.8.2.0
Rabbit-MQ	3.5.3
Apache Cassandra	2.1.5
Lightning	NA

6
Deployment

In our deployment process, we rely heavily on Docker[15] and Weave[64] for the deployment of self-contained Kafka, Rabbit-MQ, Cassandra, Lightning-viz and Spark clusters. Using Docker, we can flexibly deploy all our required modules as containerized applications. However, since each of these modules can be its-self a multi-container distributed application, we use Weave to ensure that each of these distributed applications can seamlessly communicate (together) across IP-networks.

We provide ready made Docker-images for each of our utilized technologies, as well as a set of deployment scripts that use Weave alongside Docker for single-host (local) and cluster deployments. Alternatively, for local-deployments, we

provide a single Docker-compose[16] script[1]. Although Weave is very simple and provides a complete set of features that works seamlessly with minimal configuration, it doesn't offer the best performance[63]. In this chapter, we start by giving a quick overview on Weave, then we proceed to describing how we prepared each of the provided Docker-images to run with Weave and their associated Weave commands. At the end of this chapter we provide examples for running multi-node and local clusters using Weave as well as an example for running a local cluster using Docker-compose. We assume the readers are familiar with Docker and its related concepts.

6.1 Weave Overview

Weave is a technology that works alongside Dockers existing "single host" networking capabilities[2]. It creates a virtual network that connects Docker containers deployed across multiple hosts and enables their automatic discovery. Within the weave network, each container is assigned an IP address and is accessible through this address by other containers in the network. Moreover, a single weave network can host multiple, isolated applications, with each application's containers being able to communicate with each other but not containers of other applications. This can be accomplished by assigning each application a different sub-net. If desired, a container can be attached to multiple sub-nets when it is started.

Using Weave, Docker containers could be configured to run and commu-

[1]Docker-compose is a tool for defining and running multi-container applications with Docker

[2]Along the end of our research, we have been reading about Dockers new orchestration tools that allow native multi-host networking. Among these are docker-compose [16] and docker-swarm [20] that can be stacked together to do the job. According to [17], docker-compose can be used to define the containers that comprise the distributed application and how they are connected together and through integration with docker-swarm, the multi-container application can be immediately networked across multiple hosts and can communicate seamlessly across a cluster of machines with a single command.

nicate without configuring port mappings or links. Our approach is to assign a unique IP-address to each container in the weave network, and pass that address as an input environment variable, with corresponding appropriate name, to other containers that will need to access that container. For simplicity, we attach all our containers to the same weave sub-net, so that they can all access each other. A different setting is to attach each application (ex. Kafka, Cassandra, ...etc.) to a different sub-net, while attaching the Spark driver, and workers to all of these sub-nets.

6.2 Spark Deployment

We deploy a standalone Spark cluster using a Docker-image inspired by the docker-spark project[19] available at the Spark-packages index. The docker-spark project provides a single image from which one could run a spark-master, spark-worker or a spark-shell container by executing its appropriate start up commands on running the image. Additionally, its author(s) provide a very convenient way to run a standalone Spark cluster on the localhost. We extend this to work with our Weave based multiple host deployments.

In order to start a standalone Spark cluster, each node being a master, driver or a worker needs to know its local IP-address and the master IP-address. The main idea provided by the authors of [19] is to use Docker-links to allow each worker/driver identify the master IP-address through the exposed connectivity information by the links, via environment variables[18]. Initially, each node, including the master, identifies its Docker-attached IP-address by looking at the first entry of its /etc/hosts. Worker/Driver nodes having a link to the Spark master can access the master on port 7077, the default port for Spark master and register themselves to the master. The master IP will be exposed as an environment variable at each worker/driver node as a result of the link with the master node. The environment variable name will

depend on the name of the master container.[71].

Weave gives two advantages over docker links (1) It allows containers to communicate across different hosts (2) One can assign any IP-address on the Weave network to any container, thus the IP-addresses of containers can be known in advance. We have adjusted the Dockerfile provided in [19] to work with Spark 1.4.1 instead of Spark 1.3.0 and we allow each node to look up its IP-address and the master IP-address from SPARK_LOCAL_IP and SPARK_MASTER_IP environment variables which shall be provided while running the containers.

6.2.1 Multi-node Spark Deployment Script

The following script provides an example for the deployment of a Spark master and two worker nodes using Weave.

```
#ImageName: rugdsdev/spark
#Master IP on the Weave network: 10.0.0.51
#Workers IP on the Weave network: 10.0.0.52, 10.0.0.53

weave run 10.0.0.51/24 -t -e SPARK_LOCAL_IP=10.0.0.51 rugdsdev/spark /start-master.sh

weave run 10.0.0.52/24 -t -e SPARK_LOCAL_IP=10.0.0.52 -e SPARK_MASTER_IP=10.0.0.51
rugdsdev/spark /start-worker.sh

weave run 10.0.0.53/24 -t -e SPARK_LOCAL_IP=10.0.0.53 -e SPARK_MASTER_IP=10.0.0.51
rugdsdev/spark /start-worker.sh
```

6.3 Kafka Deployment

We deploy a standalone Kafka cluster using Zookeeper and Kafka images provided in [34]. Typically Kafka is run as a cluster comprised of one or more servers and relies on Zookeeper for coordination. This requires the existence of at least a single Zookeeper instance.

The provided images allow to run a single Zookeeper, multi-broker Kafka cluster out of the box using Docker-compose. The Zookeeper image is run before any broker is alive and doesn't require any parameters or prior knowledge about the brokers. Each broker instance however needs to take as an input its docker host IP-address, a unique broker ID as well as a Zookeeper connection string, in-order to find the Zookeeper cluster and register itself. That is the minimum amount of information required by a broker to start-up and attach to the cluster properly. Additionally, any Kafka parameter can be customized by setting a corresponding environment variable having the same parameter name.

The Zookeeper and Kafka images provided in [34] where sufficient to run a single Zookeeper, multiple broker Kafka cluster that spans multiple Docker hosts using Weave. The only requirement was to provide the proper environment variables as parameters while running each Kafka broker instance.

6.3.1 Single Zookeeper, Multi-broker Kafka Deployment Script

The following script provides an example for the deployment of Kafka cluster comprising a single Zookeeper and two brokers.

```
#ZK ImageName: wurstmeister/zookeeper
#ZK IP on the Weave network: 10.0.0.11
#Broker ImageName: wurstmeister/kafka:0.8.2.0
#Brokers IP on the Weave network: 10.0.0.12, 10.0.0.13

weave run 10.0.0.11/24 -p 2181:2181 wurstmeister/zookeeper

weave run 10.0.0.12/24 -p 9092:9092 -e KAFKA_ADVERTISED_HOST_NAME="10.0.0.12" -e
KAFKA_ZOOKEEPER_CONNECT="10.0.0.11:2181" -e KAFKA_BROKER_ID=2 -e
KAFKA_AUTO_CREATE_TOPICS_ENABLE="false" -v /var/run/docker.sock:/var/run/docker.sock
wurstmeister/kafka:0.8.2.0

weave run 10.0.0.13/24 -p 9092:9092 -e KAFKA_ADVERTISED_HOST_NAME="10.0.0.13" -e
KAFKA_ZOOKEEPER_CONNECT="10.0.0.11:2181" -e KAFKA_BROKER_ID=3 -e
KAFKA_AUTO_CREATE_TOPICS_ENABLE="false" -v /var/run/docker.sock:/var/run/docker.sock
wurstmeister/kafka:0.8.2.0
```

6.4 Rabbit-MQ Deployment

We deploy a single node Rabbit-MQ broker by using the official Rabbit-MQ Docker-image provided in [48]. We do not need to provide any information concerning the host IP, on the Weave network, while running the container. However, its important to keep track of that IP since it shall be used later for accessing Rabbit-MQ.

6.4.1 Single Rabbit-MQ Broker Deployment Script

The following script provides an example for the deployment of a single broker Rabbit-MQ instance.

```
#ImageName: rabbitmq:3.5.3
#Broker IP on the Weave network: 10.0.0.21

weave run 10.0.0.21/24 -p 5672:5672 rabbitmq:3.5.3
```

6.5 Cassandra Deployment

We deploy a multiple node Cassandra cluster using the Cassandra image provided by the distributed systems group. The image is well suited to run with Weave and Weave deployment scripts were already provided on the group's Wiki.

6.5.1 Multi-node Cassandra Deployment Script

The following script provides an example for the deployment of a three node Cassandra cluster.

```
#ImageName: rugdsdev/env:cassandra-2.1.5-1-latest
#Cassandra Nodes IP on the Weave network: 10.0.0.31, 10.0.0.32, 10.0.0.33
#Cassandra Seed Nodes: 10.0.0.31, 10.0.0.32

weave run 10.0.0.31/24 -p 9042:9042 -e IP=10.0.0.31 -e SEEDS=10.0.0.31,10.0.0.32
rugdsdev/env:cassandra-2.1.5-1-latest

weave run 10.0.0.32/24 -p 9042:9042 -e IP=10.0.0.32 -e SEEDS=10.0.0.31,10.0.0.32
rugdsdev/env:cassandra-2.1.5-1-latest

weave run 10.0.0.33/24 -p 9042:9042 -e IP=10.0.0.33 -e SEEDS=10.0.0.31,10.0.0.32
rugdsdev/env:cassandra-2.1.5-1-latest
```

6.6 Lightning Deployment

We deploy a single node Lightning server using the image provided at [42]. We do not need to provide any information concerning the host IP, on the Weave network, while running the container. However, its important to keep track of that IP since it shall be used later for accessing the Lightning server.

6.6.1 Lightning Server Deployment Script

The following script provides an example for the deployment of a Lightning server.

```
#ImageName: deardooley/lightning-viz
#Lightning Server IP on the Weave network: 10.0.0.41

weave run 10.0.0.41/24 -p 3000:3000 deardooley/lightning-viz
```

6.7 Example Deployment Scripts

In this section we provide example scripts for running multi-node and local clusters using Weave as well as an example for running a local cluster using Docker-compose. For the local clusters, we do not include corresponding

commands for running Spark master, driver or workers. We assume a local Spark deployment will be used.

6.7.1 Multi-node SMART cluster using Weave

In this sub-section we provide an example script for running a multi-node SMART cluster using Weave. The cluster contains three nodes, comprising the following containers and IP-addressed on the Weave network.

Node1 [IP: 10.0.0.1]

1. Lightning Server, IP: 10.0.0.41
2. Kafka Zookeeper, IP: 10.0.0.11
3. Cassandra Node, IP: 10.0.0.31
4. Spark Master, IP: 10.0.0.51

```
#launching weave and giving the other nodes IP addresses
weave launch $Node2_IP $Node3_IP
weave expose 10.0.0.1/24

#lightning server
weave run 10.0.0.41/24 -p 3000:3000 deardooley/lightning-viz

#zk server
weave run 10.0.0.11/24 -p 2181:2181 wurstmeister/zookeeper

#cassandra node
weave run 10.0.0.31/24 -p 9042:9042 -e IP=10.0.0.31 -e SEEDS=10.0.0.31,10.0.0.32
rugdsdev/env:cassandra-2.1.5-1-latest

#spark master
weave run 10.0.0.51/24 -t -e SPARK_LOCAL_IP=10.0.0.51 rugdsdev/spark /start-
master.sh
```

Node2: [IP: 10.0.0.2]

1. Kafka Broker, IP: 10.0.0.12
2. Cassandra Node, IP: 10.0.0.32
3. Spark Worker, IP: 10.0.0.52
4. Spark Driver, IP: 10.0.0.60

```
#launching weave and giving the other nodes IP addresses
weave launch $Node1_IP $Node3_IP
weave expose 10.0.0.2/24

#kafka broker
weave run 10.0.0.12/24 -p 9092:9092 -e KAFKA_ADVERTISED_HOST_NAME="10.0.0.12" -e
KAFKA_ZOOKEEPER_CONNECT="10.0.0.11:2181" -e KAFKA_BROKER_ID=2 -e
KAFKA_AUTO_CREATE_TOPICS_ENABLE="false" -v /var/run/docker.sock:/var/run/docker.sock
wurstmeister/kafka:0.8.2.0

#cassandra node
weave run 10.0.0.32/24 -p 9042:9042 -e IP=10.0.0.32 -e SEEDS=10.0.0.31,10.0.0.32
rugdsdev/env:cassandra-2.1.5-1-latest

#spark worker
weave run 10.0.0.52/24 -t -e SPARK_LOCAL_IP=10.0.0.52 -e SPARK_MASTER_IP=10.0.0.51
rugdsdev/spark /start-worker.sh

#spark submit [DRIVER]
weave run 10.0.0.60/24 -t -e SPARK_LOCAL_IP=10.0.0.60 rugdsdev/spark
```

Node3: [IP: 10.0.0.3]

1. Kafka Broker, IP: 10.0.0.13
2. Rabbit-MQ Broker, IP: 10.0.0.21
3. Spark Worker, IP: 10.0.0.53
4. Spark Worker, IP: 10.0.0.54

```
#launching weave and giving the other nodes IP addresses
weave launch $Node1_IP $Node2_IP
weave expose 10.0.0.3/24

#kafka broker
weave run 10.0.0.13/24 -p 9092:9092 -e KAFKA_ADVERTISED_HOST_NAME="10.0.0.13" -e
KAFKA_ZOOKEEPER_CONNECT="10.0.0.11:2181" -e KAFKA_BROKER_ID=3 -e
KAFKA_AUTO_CREATE_TOPICS_ENABLE="false" -v /var/run/docker.sock:/var/run/docker.sock
wurstmeister/kafka:0.8.2.0

#Rabbit-MQ broker
weave run 10.0.0.21/24 -p 5672:5672 rabbitmq:3.5.3

#2xspark worker
weave run 10.0.0.53/24 -t -e SPARK_LOCAL_IP=10.0.0.53 -e SPARK_MASTER_IP=10.0.0.51
rugdsdev/spark /start-worker.sh
weave run 10.0.0.54/24 -t -e SPARK_LOCAL_IP=10.0.0.54 -e SPARK_MASTER_IP=10.0.0.51
rugdsdev/spark /start-worker.sh
```

6.7.2 Local SMART cluster using Weave Deployment Script

```
weave launch
weave expose 10.0.0.1/24

weave run 10.0.0.11/24 -p 2181:2181 wurstmeister/zookeeper
weave run 10.0.0.12/24 -p 9092:9092 -e KAFKA_ADVERTISED_HOST_NAME="10.0.0.12" -e
KAFKA_ZOOKEEPER_CONNECT="10.0.0.11:2181" -e KAFKA_BROKER_ID=2 -e
KAFKA_AUTO_CREATE_TOPICS_ENABLE="false" -v /var/run/docker.sock:/var/run/docker.sock
wurstmeister/kafka:0.8.2.0

weave run 10.0.0.21/24 -p 5672:5672 rabbitmq:3.5.3

weave run 10.0.0.31/24 -p 9042:9042 -e IP=10.0.0.31 -e SEEDS=10.0.0.31
rugdsdev/env:cassandra-2.1.5-1-latest

weave run 10.0.0.41/24 -p 3000:3000 deardooley/lightning-viz
```

6.7.3 Local SMART cluster using Docker-Compose

```
rmqBrokerLocal:
  image: rabbitmq:3.5.3
  ports:
    - "5672:5672"
  environment:
    RABBITMQ_NODENAME: "my-rabbit"
cassandraLocal:
  image: rugdsdev/env:cassandra-2.1.5-1-latest
  ports:
    - "9042:9042"
    - "7199:7199"
  environment:
    IP: "127.0.0.1"
    JMX_PASSWORD: "12345"
kafkaZookeeperLocal:
  image: wurstmeister/zookeeper
  ports:
    - "2181:2181"
kafkaBrokerLocal:
  image: wurstmeister/kafka:0.8.2.0
  ports:
    - "9092:9092"
  links:
    - kafkaZookeeperLocal:zk
  environment:
    KAFKA_ADVERTISED_HOST_NAME: "127.0.0.1"
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
lightningLocal:
  image: deardooley/lightning-viz
  ports:
    - "3000:3000"
```

7
Conclusion

The Internet of Things (IoT) is transforming all the objects around us into an ecosystem of information that will enrich our lives. While IoT represents the convergence of advances in miniaturization, wireless connectivity, increased data storage capacity and batteries, the IoT would not be possible without sensors. Sensors detect and measure changes in position, temperature, light, etc. and they are necessary to turn billions of objects into data-generating "things" that can report on their status, and in some cases, interact with their environment.

In this project we demonstrated the use of sensor-data in the domain of environmental monitoring and actuation. We have provided a system, called SMART, that is capable of collecting and analyzing massive volumes of time-

series data generated from sensors, to get the clearest picture to predict and forecast future environmental conditions. Though there is a lot of excitement about big data analytics, it was very challenging to efficiently fit big data technologies into their technology stack and put it to use in a practical application. We have applied and tested our system on a simulated HVAC domain, where a set of rooms equipped with sensors have been monitored, analyzed and queried to regulate their individual temperature levels.

Bibliography

[1] 0xdata. http://0xdata.com/.

[2] Aerosolve. http://spark-packages.org/package/airbnb/aerosolve.

[3] Airbnb, a website for people to list, find, and rent lodging. https://www.airbnb.com/.

[4] The apache cassandra project. http://cassandra.apache.org/.

[5] Apache flume. https://flume.apache.org/.

[6] Apache kafka - a high-throughput distributed messaging system. http://kafka.apache.org/.

[7] Apache samza. http://samza.apache.org/.

[8] The apache software foundation announces apache flink as a top-level project. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69,.

[9] The apache software foundation announces apache spark as a top-level project. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50,.

[10] The apache software foundation announces apache storm as a top-level project. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces64,.

[11] Apache spark - lightning-fast cluster computing. https://spark.apache.org/.

[12] A brief introduction to apache cassandra. https://academy.datastax.com/demos/brief-introduction-apache-cassandra.

[13] A community index of packages for apache spark. http://spark-packages.org/.

[14] Distributed rabbit-mq brokers. http://www.rabbitmq.com/distributed.html,.

[15] Docker, an open platform for distributed applications for developers and sysadmins. https://www.docker.com/.

[16] Docker compose. https://docs.docker.com/compose/.

[17] Docker delivers native multi-host networking to advance distributed application portability. https://www.docker.com/docker-news-and-press/06.22.2015_Docker-Delivers-Native-Multi-Host-Networking-to-Advance-Distributed-Applicati

[18] Docker links. https://docs.docker.com/userguide/dockerlinks/.

[19] docker-spark. http://spark-packages.org/package/epahomov/docker-spark.

[20] Docker swarm. https://docs.docker.com/swarm/.

[21] Extending rdds for fun and profit. http://rahulkavale.github.io/blog/2014/12/01/extending-rdd-for-fun-and-profit/.

[22] Extending sparks api. http://blog.madhukaraphatak.com/extending-spark-api/.

[23] Facebook. http://facebook.com/.

[24] Facebook, inbox search. https://www.facebook.com/notes/facebook/inbox-search/20387467130.

[25] Facebook's cassandra paper, annotated and compared to apache cassandra 2.0. http://docs.datastax.com/en/articles/cassandra/cassandrathenandnow.html.

[26] Flink: Data streaming fault tolerance. https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html.

[27] Getting started with time series data modeling. http://planetcassandra.org/getting-started-with-time-series-data-modeling/.

[28] Git hub: Apache flink. https://github.com/apache/flink/,.

[29] Git hub: Apache spark. https://github.com/apache/spark,.

[30] H2o. http://h2o.ai/.

[31] Hbase. http://hbase.apache.org/.

[32] Hmmi janelia resaerch campus. http://janelia.org/. Accessed: 2015-4-01.

[33] The internet of things: A time series data challenge. http://www.ibmbigdatahub.com/blog/internet-things-time-series-data-challenge.

[34] kafka-dcker. https://github.com/wurstmeister/kafka-docker.

[35] Kafka-spark-consumer: Low level kafka-spark consumer. http://spark-packages.org/package/dibbhatt/kafka-spark-consumer.

[36] killrweather. http://spark-packages.org/package/killrweather/killrweather.

[37] Kryo serialization. https://github.com/EsotericSoftware/kryo.

[38] Lightening, data visuzliation server. http://lightning-viz.org/.

[39] Lightning - data visualization sserver. http://lightning-viz.org/.

[40] Lightning data visualization server. https://github.com/lightning-viz/lightning.

[41] Lightning-scala api. https://github.com/lightning-viz/lightning-scala.

[42] Lightning-viz docker. https://github.com/lightning-viz/lightning.

[43] meetup-stream. http://spark-packages.org/package/actions/meetup-stream.

[44] Meetup.com. http://www.meetup.com/.

[45] Mongo db official site. www.mongodb.com.

[46] Pull request: Lightning-scala api. https://github.com/lightning-viz/lightning-scala/pull/11.

[47] Rabbitmq - messaging that just works. https://www.rabbitmq.com/.

[48] Rabbitmq docker. https://registry.hub.docker.com/_/rabbitmq/.

[49] Rabbitmq-receiver. http://spark-packages.org/package/Stratio/RabbitMQ-Receiver.

[50] Real-time analytics at facebook. http://www-conf.slac.stanford.edu/xldb11/talks/xldb2011_tue_0940_FacebookRealtimeAnalytics.pdf.

[51] Sort benchmark home page. http://sortbenchmark.org/,.

[52] Spark cassandra connector, connects spark to cassandra. http://spark-packages.org/package/datastax/spark-cassandra-connector.

[53] Spark-kafka: Low level integration of spark and kafka. http://spark-packages.org/package/tresata/spark-kafka.

[54] spark-ml-streaming. http://spark-packages.org/package/freeman-lab/spark-ml-streaming.

[55] Spark streaming + kafka integration guide. http://spark.apache.org/docs/latest/streaming-kafka-integration.html,.

[56] Sparkling water. http://spark-packages.org/package/h2oai/sparkling-water.

[57] Stratio: Big data, pure spark. http://www.stratio.com/.

[58] Stratio deep, connecting apache spark with different data stores. http://spark-packages.org/package/Stratio/deep-spark.

[59] Stream processing explained. http://www.sqlstream.com/stream-processing/.

[60] Streaming cep engine. http://spark-packages.org/package/Stratio/streaming-cep-engine.

[61] "time series analysis". basic statistics and data analysis. http://itfeature.com/time-series-analysis-and-forecasting/time-series-analysis-forecasting.

[62] Ting zhu, sheng xiao, qingquan zhang, yu gu, ping yi, and yanhua li, "emergent technologies in big data sensing: A survey," international journal of distributed sensor networks, article id 902982, in press.

[63] Weave is kinda slow. `http://www.generictestdomain.net/docker/weave/networking/stupidity/2015/04/05/weave-is-kinda-slow/`.

[64] Weave, run your applications containers anywhere without compromises. `http://weave.works/`.

[65] Zen. `http://spark-packages.org/package/cloudml/zen`.

[66] Yifan Bo and Haiyan Wang. The application of cloud computing and the internet of things in agriculture and forestry. In *Service Sciences (IJCSS), 2011 International Joint Conference on*, pages 168–172, May 2011.

[67] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows. *ArXiv e-prints*, June 2015.

[68] Davies Liu Cody Koeninger and Tathagata Das. Improvements to kafka integration of spark streaming. `https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html`,.

[69] L. Coetzee and J. Eksteen. The internet of things - promise for the future? an introduction. In *IST-Africa Conference Proceedings, 2011*, pages 1–9, May 2011.

[70] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[71] E.Pahomov.

[72] Jeremy Freeman. Analytics + visualization for neuroscience: Spark, thunder, lightning.

[73] Jeremy Freeman, Nikita Vladimirov, Takashi Kawashima, Yu Mu, Nicholas J. Sofroniew, Davis V. Bennett, Joshua Rosen, Chao-Tsung Yang, Loren L. Looger, and Misha B. Ahrens. Mapping brain activity at scale with cluster computing. *Nature Methods*, July 2014.

[74] E. Frenkiel. Real time trinity. `http://thenewstack.io/apache-kafka-spark-database-real-time-trinity/`.

[75] D. Gachet, M. de Buenaga, F. Aparicio, and V. Padron. Integrating internet of things and cloud computing for health services provisioning: The virtual cloud carer project. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 918–921, July 2012.

[76] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[77] A Christy Persya Lakshmi K K, Girija N C. Integration of cloud computing for iot. *International Journal of Emerging Research in Management Technology*, 4(5):371 – 375, 2015.

[78] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[79] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78, Dec 2014.

[80] A. Mostosi. Big-data ecosystem: Incomplete but useful list of big-data related project. https://github.com/zenkay/bigdata-ecosystem.

[81] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand 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.

[82] Michael G. Noll. Integrating kafka and spark streaming: Code examples and state of the game. http://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/.

[83] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys 2013*. ACM, April 2013. Updated version.

[84] B.B.P. Rao, P. Saluia, N. Sharma, A. Mittal, and S.V. Sharma. Cloud computing for internet of things amp; sensing based applications. In *Sensing Technology (ICST), 2012 Sixth International Conference on*, pages 374–380, Dec 2012.

[85] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "all roads lead to rome": optimistic recovery for distributed iterative data processing. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, CIKM '13, pages 1919–1928, New York, NY, USA, 2013. ACM.

[86] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[87] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[88] Y. Ahmad H. Balakrishnan M. Balazinska M. Cherniack J. Hwang W. Lindner S. Madden A. Maskey A. Rasin E. Ryvkina M. Stonebraker N. Tatbul Y. Xing S. Zdonik. U. Cetintemel, D. Abadi. *The Aurora and Borealis Stream Processing Engines: Book chapter in Data Stream Management: Processing High-Speed Data Streams, edited by M. Garofalakis, J. Gehrke, R. Rastogi.* Springer, 2007.

[89] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[90] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[91] Wenying Zeng, Chao Huang, Banxiang Duan, and Fagen Gong. Research on internet of things of environment monitoring based on cloud computing. In *Automatic Control and Artificial Intelligence (ACAI 2012), International Conference on*, pages 1724–1727, March 2012.