Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Computer Science	Spring semester 2015 Open access
Writer: **Maziar Kaveh**	
Faculty supervisor: **Tomasz Wiktorski**	
Title of thesis: **ETL and Analysis of IoT data using OpenTSDB, Kafka, and Spark**	
Credits (ECTS): 30	
Key Words: **Time-series database; IoT; HBase; Apache Kafka; OpenTSDB; Spark; Machine learning**	Pages: 63 Stavanger, 15/06/2015

ETL and Analysis of IoT data using OpenTSDB, Kafka, and Spark

Maziar Kaveh

Faculty of Science and Technology

University of Stavanger

Jun 2015

Abstract

The Internet of Things (IoT) is becoming increasingly prevalent in today's society. Innovations in storage and processing methodologies enable the processing of large amounts of data in a scalable manner, and generation of insights in near real-time. Data from IoT are typically time-series data but they may also have a strong spatial correlation. In addition, many time-series data are deployed in industries that still place the data in inappropriate relational databases.

Many open-source time-series databases exist today with inspiring features in terms of storage, analytic representation, and visualization. Finding an efficient method to migrate data into a time-series database is the first objective of the thesis.

In recent decades, machine learning has become one of the backbones of data innovation. With the constantly expanding amounts of information available, there is good reason to expect that smart data analysis will become more pervasive as an essential element for innovative progress. Methods for modeling time-series data in machine learning and migrating time-series data from a database to a big data machine learning framework, such as Apache Spark, is explored in this thesis.

Keywords-component; Time-series database; IoT; HBase; Apache Kafka; OpenTSDB; Spark; Machine learning

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisors Associate Professor Tomasz Wiktorski, for the continuous support of my thesis work, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

My sincere thanks also goes to Assistant Professor. Khosro Fazli, for his helps and supports, during research on machine learning.

I would like to thank my friends Maryam Mareshi and Long Cui for their inspirations and help.

Last but not the least, I would like to thank my family: my family and friends for supporting me spiritually throughout my life.

Maziar Kaveh University of Stavanger

Contents

List of Tables and Equations

Chapter 1

Introduction

1.1 Background

In recent years, the Internet has expanded exponentially from a small research network, containing just a couple of nodes, to an extensive worldwide network that services more than one billion users. Scaled down size and cost decreases of electronic gadgets makes it conceivable that the Internet can expand into another context: smart items, i.e., ordinary physical items that are upgraded with a small device to provide local intelligence and connectivity. The small electronic device, a data processing component that is attached to a physical thing, connects the physical world and the information world. A smart object is a physical system or an embedded system, consisting of a thing (the physical entity) and a component that processes the sensor data and supports a wireless communication link to the Internet.

Kevin Ashton initially instituted the term 'Internet of Things' in 1999 within the context of chain management. However, in the last decade, the definition has expanded to cover an extensive variety of uses such as health awareness, utilities, transport, and others. Although the meaning of "Things" has changed as technology advanced, the primary objective remains unchanged: having a computer acquire information without the need for human interaction.

The challenge of the IoT (Internet of Things) is not in the functionality of a smart object, but in the extreme number of billions or even trillions of smart objects that create novel technical issues related to large amounts of data. Such data, called time series data, follow events or repeated estimations as an element of time, generating a sequence of observations on a variable measured at progressive points in time. Data may be collected progressive points in time. The pattern of the data provides important information regarding how the time series has behaved previously. Where such a pattern can be expected to continue in the future, we can use past behavior to guide us in selecting an appropriate forecasting method.

A general example of the IoT is wireless sensors. A recent advance in the field of

Micro Electro Mechanical Systems (MEMS), low-power microelectronics and low-power communication has made it possible to assemble small integrated smart objects, called sensor nodes, containing a sensor, a micro-controller and a wireless communication controller. A sensor node can acquire a mixture of physical, chemical, or biological signals to measure properties of its environment. Sensor nodes have limited resources: they are powered either by a small battery or by energy harvested from the environment; they have constrained computational power; little memory; and few communication capabilities.

Given that the end goal is to monitor and observe some event or metric, anywhere from tens to millions of sensor nodes are deployed systematically or randomly in a sensor field to form an ad-hoc self-organizing network – a wireless sensor network (WSN). The WSN collects data about the targeted phenomenon and transmits the data via an ad-hoc multi-hop communication channel to one or more base stations that can be connected to the Internet.

Until recently, the standard way to deal with managing large-scale time series data has been to choose prior to analysis which data to sample, to study a portion of the sampled data, deliver the desired reports, summarize a few outcomes to be archived, and then dispose of most of the raw data. Currently, the opportunity to do more extensive and profound examinations of the data is evolving, allowing the investigation of information that would previously have been discarded. At present-day rates of data generation, even a couple of weeks or months of data is of sufficiently large volume to overpower customary database techniques. However, the new scalable NoSQL platforms and tools for data storage and access make it possible to document years of raw or lightly processed data. These finer-grained and longer histories are particularly profitable for providing the information required for forecasting, oddity discovery, back-testing new models, and discovering long-term trends and correlations.

As a consequence of these new smaller and cheaper processors technologies, the frequency of circumstances in which data are being gathered as time series is expanding, as are the requirement for reliable, high-performance time series databases. Note that it is not simply an issue of what data to save, but rather when considering what data maintain, a time series database is profitable. At extensive scales, time-based queries can be implemented as large, contiguous scans that are extremely effective if the information is stored appropriately in a time series database. In addition, if the amount of information is vast, a non-relational time

series database (TSDB) in a NoSQL system can be expected to give adequate scalability.

There are many time-series data deployed in industries that are still placing the data in relational databases.

There are three main objectives of this thesis. The first is finding an efficient method to migrate data into a time-series database. The second is to make a platform to stream time series data from a TSDB into a cluster-computing framework. The third is to use machine-learning approaches for analysis, modeling and forecasting of data.

As an example, data from sensors deployed at KE-HUS (as part of EU FP7 SEEDS project) are used. The data nature and structure are described in chapter 4

1.2 Thesis Organization

The thesis is organized as follows.

Chapter 2 provides basic background theory needed to understand the thesis.

Chapter 3 has a brief introduction to machine learning for time series data and then describes how to use spark for forecasting the data.

Chapter 4 describes the design of thesis implementation and how to import data into the OpenTSDB.

Chapter 5 is a plugin developed for OpenTSDB that pushes real-time data into Apache Kafka.

Chapter 6 describes tests of the framework.

Chapter 7 concludes the thesis, and indicates some planned future enhancements.

CHAPTER 2

Background

2.1 Big Data

Within the past decade, the amount of information being generated is more than 20 terabytes per second and continues to expand. Volume and speed, as well as the admixture of material from organized to semi-organized in nature, implies that the information is originating from blog entries, tweets, informal organizational communications, photographs, videos, permanently created log messages about what clients are doing, etc. Consequently, big data is a mix of transactional information and intuitive information. This extensive set of information is further utilized by organizations for decision making. Recording, analyzing, and outlining these vast datasets proficiently and at a viable cost is likely to prove challenging for organizations needing to collect such data.

In 2003, Google distributed a paper on a scalable distributed file system called Google File System (GFS), which utilizes a group of commodity hardware to store tremendous amounts of information and guarantee high accessibility to it by through replication of information between nodes. Later, Google distributed an extra paper on handling extensive, distributed datasets using MapReduce (MR).

To handle big data, platforms, for example, Hadoop, which inherits the essentials from both GFS and MR, were created and made available to the community. A Hadoop-based platform has the capacity to store, and process, constantly developing information in terabytes or petabytes.

Four main features of NoSQL, apply to most NoSQL databases. These features are listed below, and NoSQL is contrasted with legacy relational DBMS:

- **Schema agnostic**: A database schema is the description of all possible information and data structures in a relational database. With a NoSQL database, a schema is not required, giving you the flexibility to store data

without doing up-front schema design.

- **Non-relational**: Relations in a database set up associations between tables of information. For instance, a list of transaction details could be connected to a different list of delivery details. With a NoSQL database, this data is stored as a whole — a solitary record with everything about the transaction, including the delivery address.
- **Commodity hardware**: Some databases are intended to work best (or only) with particular storage and processing hardware. With a NoSQL database, inexpensive off-the-shelf servers can be used. The ability to have more of these cheap servers allows NoSQL databases to scale to handle more information.
- **Profoundly distributable**: Distributed databases can store and process an arrangement of data on more than one device. With a NoSQL database, a group of servers can be utilized to hold a single huge database.

2.1.1 HBase

HBase is an Apache open source project implementation of Google's BigTable that provides Big Table storage capabilities for Hadoop. Data are logically organized into tables, rows and columns. Columns in HBase can have multiple versions of the same row key. The data model is similar to that of Big Table. Data are replicated across a number of nodes.

Each table must have an element defined as a primary key, and all access attempts to HBase tables must use this primary key.

A typical HBase cluster has one active master, one or several backup masters, and a list of regional servers.

- **HBaseMaster**: The HBaseMaster is responsible for assigning regions to HRegionServers. The first region is the ROOT region, which contains all the META regions to be assigned. It also monitors the health of HRegionServers and, if it detects a failure in HRegionServer recover it using replicated data. Furthermore, the HBaseMaster is responsible for maintenance of the table, performing such tasks as on-/off-lining of tables and changes to table schema - adding and removing column families, etc.
- **HRegionServer**: The HRegionServer serves client read and write requests. It interacts with the HBaseMaster to obtain a list of regions to serve and to

inform the master that it is working.

- **HBase Client:** The HBase client is responsible for investigating HRegionServers that serve the specific row range of interest. On interaction, the HBase client communicates with the HBaseMaster to find the location of the ROOT region.

2.1.2 Apache Cassandra

Apache Cassandra, first developed at Facebook and built on Amazon's Dynamo and Google's BigTable, is a distributed storage system for managing large numbers of structured data across many commodity servers, providing high availability with no single point of failure. Cassandra can run on top of an infrastructure of hundreds of nodes. While Cassandra is like a database and shares many design and implementation strategies of database systems, it does not fully support a relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format.

Key features of Apache Cassandra are

- **Elastic scalability** - More space for data and client support can be reached by easily adding new nodes.
- **Always on architecture** - There is no single point of failure resulting in continuous availability for applications that cannot afford to lose aliveness.
- **Fast linear-scale performance** - By increasing the number of nodes, response times will grow linearly.
- **Flexible data storage** - structured, semi-structured and unstructured data format could be supported effortlessly.

Transaction support - supports atomicity, isolation and durability of ACID (Atomicity, Consistency, Isolation, and Durability) with tunable consistency.

There are many open source and commercial implementations of time series databases and each is customized to meet a specific need. Here, I needed an open source TSDB that relies on big table implementation and supports the functionality for query data based on inserting metadata.

OpenTSDB, KairosDB, InfluxDB and Druid are discussed.

2.2 Open-source Time Series Databases

The TSDB is used to insert, enumerate, update and delete various time series data and organize them. The inserted data might be organized hierarchically and have metadata available with them. The server often supports sets of aggregation functions such as sum, min, max, avg, etc. to aggregate data over some period of time. OpenTSDB is able to filter them over the metadata available on the inserted row.

In order to store and retrieve data in a TSDB, a distributed NoSQL database management system (DBMS) with high scalability and elasticity is needed. The DBMS must be able to store billions of pieces of data and continuously adjust to changes in workload to efficiently use hardware resources available. The DBMS should also be optimized for write operations and support incidental read operations. Two examples of such a DBMS are Cassandra and HBase.

2.2.1 OpenTSDB

The Open Source, Scalable, Time Series Database (OpenTSDB) stores and serves massive amounts of time series data without losing granularity. There is no master, nor shared state in the architecture. OpenTSDB uses HBase to store and retrieve time-series data. The schema for HBase is highly optimized for fast aggregations and minimal storage space. HBase does not require direct communication; it can be accessed via a telnet-style protocol, an HTTP API or a simple built-in GUI, which offers a comprehensive web interface for querying, displaying and analyzing the time series data.

2.2.2 KairosDB

KairosDB is a rewrite version of OpenTSDB that more clearly separates data retrieval from data representation. Therefore, KairosDB also includes more query functionality, such as more advanced time series aggregators. KairosDB uses Cassandra as a storage layer for improved speed and scalability. However, an abstraction at the storage layer allows using other DBMS's also. HBase, with some

limitations, was supported in earlier versions, however it is no longer supported. Originally forking the OpenTSDB code base and making it into a data storage plugin for KairosDB achieved HBase support. The functionality of KairosDB is beyond what HBase code can support. Like OpenTSDB, KairosDB supports telnet and REST API as well as compressed data uploads of batch data.

2.2.3 InfluxDB

InfluxDB is a horizontally scalable time series, metrics, and analytics database. It is written in Go, has no external dependencies and features an SQL-like query language.

InfluxDB is targeted to use cases for DevOps, metrics, sensor data, and real-time analytics.

HTTP API and built-in management interface is enabled for handling data.

2.2.4 Druid

Druid is an open source data store for implementing real-time analytics on massive data sets. The framework has a column-oriented storage layout and an advanced indexing structure.

To address complex data analysis problems, different node types are declared.

The node types that currently exist are:

- **Historical nodes** handle storage and querying on historical data (not real-time)
- **Real-time nodes** are responsible for data in real-time. They are available for listening to a stream of incoming data and making it available immediately inside the Druid system.
- **Coordinator nodes** track the grouping of historical nodes to ensure that data is live and replicated.
- **Broker nodes** are interfaces that get queries from clients and send those queries to real time and historical nodes.
- **Indexer nodes** control loading batch and real-time data into the system, and allow alterations to the data stored in the system.

-

This separation allows each node type to specialize in its particular task.

Table 1 indicates the default-enabled features of four DBMS's that are discussed; note that more features might be applied to them by installing plugins.

	OpenTSDB	**KairosDb**	**Druid**	**InfluxDB**
Current version	2.0.1	0.9.3	0.6.160	0.8.8
Data store	HBase	Cassandra	In-memory OLAP	Built-in
Support grouping	x	x	x	x
Http API	x	x	x	x
Web interface	x	x	-	x
Batch import	x	x	x	-
Telnet	x	x	-	-

Table 1 Comparing TSDB

2.3 Machine learning

2.3.1 What is machine learning?

Machine learning is the exploration of having computers learn by example, without being unequivocally programmed. In previous years, auto-driving cars, speech recognition, successful web searches, and an enhanced comprehension of the human genome, have all been achieved with machine learning. Machine learning is

prevalent in technology today and it is likely that most people use it many times each day without knowing it. Numerous analysts suggest it is the optimal approach to gaining human-level artificial intelligence (AI).

Machine learning lies at the intersection of software engineering, engineering, and statistics. It is used in numerous fields from politics to geosciences. Any field that needs to decipher and follow-up on information can benefit from machine learning strategies.

Machine learning uses statistics. To many people, statistics are an impenetrable subject used by companies to promote their products deceptively. Engineering is applying science to resolve an issue. Engineers are accustomed to tackling a deterministic issue, where the answer resolves the issue immediately. However, there are numerous issues where the problem is not deterministic. That is, we do not know enough about the issue or do not have enough computing power to legitimately demonstrate the issue. For these issues, we require statistics. For instance, how to inspire people is an issue of great interest that has so many variables involved it impossible to model.

We have a massive amount of human-generated information on the Internet, but recently more non-human sources of information are impeding on the web. The innovation behind the sensors is not new; however interfacing them to the web is new. It has been estimated that physical sensors represent 20 percent of non-video web traffic.

2.3.2 Types of Learning

2.3.2.1 Supervised Learning

Supervised learning is learning in which information accompanies extra attributes that need to be predicted. Face recognition is an example, where cases must be labeled unequivocally as to whether the face is present or not. In an unsupervised calculation, samples are not labeled, i.e. no information is provided. Obviously, in such a case the calculation itself cannot invent what a face is, yet it might group the information into distinctive classes, e.g. the program might recognize that faces are altogether different from scenes, which are in turn altogether different from animals.

Supervised learning is the machine learning assignment of deriving a capacity from labeled training information. The training information comprises of an arrangement of training examples. Each sample is a couple comprising of an input instance (regularly a vector) and a desired yield value, which is called the supervisory sign.

Supervised learning is divided into two categories:

- **Classification**: Tests have two or more classes and effectively labeled information is used to acquire the ability to foresee the class of unlabeled information. An example of this is handwritten digit recognition, in which the point is to allocate every data array to one of a limited number of discrete classifications. Another approach to consider arrangement is as a discrete (rather than continuous) manifestation of directed realizing where one has a set number of classifications and for each of the n tests gave, one is to attempt to mark them to the right category or class.
- **Regression**: Where the output yield is comprised of one or more consistent variables, then the task is called regression. For example, the length of a salmon can be forecast as a component of its age and weight. Regression will be explored further later in this thesis.

2.3.2.2 Unsupervised Learning

Unsupervised learning has preparation information comprised of an arrangement of information vectors, but no corresponding target values. One possible objective here is to find groupings of comparative cases in the data, called clustering. A second possible objective is to focus the dissemination of information in the relevant data space, called density estimation. The third possible objective of unsupervised learning is to reduce the information from a high-dimensional space down to three-dimensions with the end goal of visualization.

2.3.3 Regression

A regression utilizes the chronicled relationship between an independent and a dependent variable to anticipate the future estimations of the dependent variable.

Organizations use regression to anticipate such things as future deals, stock costs, currency exchange rates, and efficiency increases resulting from a training program.

2.3.3.1 Types of Regression

Regression models use the past relationship between variables to anticipate their future conduct. As an example, consider an organization that needs to see how past advertising consumption related to sales, with the end goal of making informed future choices about promotion. The dependent variable in this example is sales and the independent variable is advertising consumption.

Typically, more than one independent variable impacts the dependent variable. For example, in the above case of sales being affected by advertising, additional variables may also affect sales: the number of offers agents and the commission rate paid to deals delegates. When only one independent variable is utilized as a part of a regression, it is known as a basic regression. When two or more independent variables are utilized, it is called a multiple regression.

Regression models can be either linear or nonlinear. A linear model assumes connections between variables are straight-line connections, while a nonlinear model allows the connections between variables to have curved lines. In business, the relationship between the arrival of an individual stock and the profits of the business sector is typically portrayed as a direct relationship, while the relationship between the cost of an item and the interest for it, is regularly displayed as a nonlinear relationship.

As should be obvious, there are a few unique classes of regression systems, with each having different degrees of complexity and logical power. The most fundamental sort of regression is the straightforward linear regression. A basic linear regression utilizes a single independent variable, and it depicts the relationship between the independent variable and dependent variable as a straight line.

2.4 Apache Spark

Spark is an innovative cluster-computing framework that is able to execute programs up to 40 times faster than Hadoop. Spark keeps MapReduce's linear scalability and fault tolerance; furthermore, it is extended in a few important ways. First, instead of depending on an inflexible map-then-reduce format, its engine can execute a more general directed acyclic graph (DAG) of operators. This means that, where MapReduce must write out intermediate results of the distributed file system, Spark can send them directly to the next step in the pipeline. In this way, like Dryad, an earlier version of MapReduce that began at Microsoft Research, Spark supplements its ability with a rich set of changes that allow users to express computation more compactly. It has a solid developer focus and a streamlined API that can represent complex pipelines in just a couple lines of code.

Finally, Spark improves its predecessors with in-memory processing. Its Resilient Distributed Dataset (RDD) abstraction gives developers the ability to emerge any point in a processing pipeline into memory over the cluster, implying that future steps that need to use same data will not have to recompute values, or fetch them from a disk. This ability handles a number of scenarios that distributed processing engines could not previously tackle.

Spark is appropriate for iterative algorithms that require multiple steps passing over a dataset and for responsive applications that rapidly react to client queries by scanning large in-memory datasets.

As well as making cluster applications faster, Spark provides a more convenient way to write code, through a brief dialect language-integrated programming interface in Scala, a popular functional language for the Java Virtual Machine (JVM). In addition, Spark is compatible with nearly any storage system supported by Hadoop, able to read data to or write data from them. This allows it to cooperate with the formats regularly used to store data on Hadoop, such as Avro and CSV, also to NoSQL databases like HBase and Cassandra.

The stream-processing library Spark Streaming is able to acquire information continuously from frameworks like Flume and Kafka. Its SQL library, SparkSQL is Spark's bundle for working and querying over structured data via SQL as well as the Apache Hive Query Language (HQL), it also supports many different sources of data like Hive tables, Parquet, and JSON. Not only Does Spark SQL support a

SQL interface, but it allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all inside a single application, in this way consolidating SQL with complex analytics.

Spark includes a library containing common machine learning (ML) functionality, called MLlib. MLlib gives numerous sorts of machine learning calculations like classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import. It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm.

All of these methods are designed to scale out across a cluster. They implement many common machine learning and statistical algorithms to simplify large-scale machine learning pipelines. For example, GraphX is a library for manipulating graphs and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs and a library of common graph algorithms.

CHAPTER 3

Forecasting time series data using machine learning

In order to forecast time series it is necessary to know the different components of time series data. In this chapter, first there is an introduction for various components of time-series, then as an example of implementation of simple linear regression, in this example only one variable, which is temperature time series data in Seed database for room E_423B is used.

In the last section, we use Apache Spark Mlib for calculated same data for but having multiple variables (inside temperature, inside humidity, outside temperature and outside humidity).

3.1 Components of a Time Series

Trends, cyclical variation, seasonal variation and irregular variation are four types of components of time series data.

3.1.1 Secular Trend

A long-term increase or decrease in the data indicates a trend. It does not need to be straight. Some trends move steadily upwards, others decrease, and still others are relatively level over the long run. It also might go from an increasing trend to a decreasing trend.

3.1.2 Cyclical Variation

The second component of a time series is the cyclical variation that happens when any pattern demonstrating an up and down movement around a given trend is recognized. The span of a cycle depends on the kind of business or industry being investigated.

3.1.3 Seasonal Variation

Seasonality happens when the time series displays consistent fluctuations during the same month (or months) every year, or during the same quarter consistently. Seasonality is dependably of a settled and known period.

3.1.4 Irregular Variation

This component is unpredictable. Every time series has some eccentric component that makes it an arbitrary variable. In the forecast, the goal is to model all the components to the point that the main component that remains unexplained is the random component.

3.2 Smoothing the time series

When we work with certifiable data, we frequently discover noise, which is characterized as pseudo-random fluctuations in values that are not connected to the observed data.

With a specific end goal of removing noise, we can utilize diverse methodologies. For example, expanding the measure of information by the insertion of new values where the series is sparse. However, as a rule this is not possible. Another methodology is smoothing the data, ordinarily done using normal or exponential strategies. The "average method" helps us to smooth time series data by supplanting each element in the series with either a simple or a weighted average of the data around it. We will limit a smoothing window to the interval of conceivable values, which controls the smoothness of the result. The primary disadvantage of utilizing the moving average methodology is that where anomalies or unexpected deviations in the original time series occur, the outcome may be incorrect and can create barbed curves.

3.3 Simple linear regression

When looking at the relationship between a quantitative result and a single quantitative variable, simple linear regression is the most often considered examination technique. The basic part lets us know we are just considering a solitary illustrative variable. We propose a linear relationship between the populace mean of the result and the estimation of the informative variable. On the off chance that we let Y be some result, and x be some logical variable, then we can express the auxiliary model utilizing the mathematical statement

$$E(Y\,|x) = \beta 0 + \beta 1 x$$

Equation 1 Linear equation

Where E(), which is read "expected value of", demonstrates a population mean; Y|x, which is called "Y given x", demonstrates that we are estimating Y when x is confined to some single quality; β0, read "beta zero", is the intercept parameter; and β1, read "beta one" is the slope parameter in the linear regression.

As an example of simple linear regression the temperature time series data of room number E_423B is used. The sample data is the month of May 2015 and for smoothing data, average of temperature data for every 6 hours is calculated. The data are shown below in JSON format.


```
{
   "2015-05-01": {"0-6": 25, "12-18": 26, "18-24": 25, "6-12": 26},
   "2015-05-02": {"0-6": 25, "12-18": 26, "18-24": 26, "6-12": 26},
   "2015-05-03": {"0-6": 26, "12-18": 25, "18-24": 26, "6-12": 25},
   "2015-05-04": {"0-6": 23, "12-18": 25, "18-24": 26, "6-12": 25},
   "2015-05-05": {"0-6": 22, "12-18": 24, "18-24": 26, "6-12": 23},
```

```
.
.
.
   "2015-05-28": {"0-6": 26, "12-18": 26, "18-24": 25, "6-12": 22},
   "2015-05-29": {"0-6": 23, "12-18": 22, "18-24": 22, "6-12": 23},
   "2015-05-30": {"0-6": 24, "12-18": 23, "18-24": 25, "6-12": 22},
   "2015-05-31": {"0-6": 23, "12-18": 23, "18-24": 24, "6-12": 25}
}
```


3.3.1 Least-Squares Regression Line

In order to calculate β0 and β1 the method of **least squares** is a standard approach in regression analysis. Equation 2 shows how to calculate.

$$E(Y\,|x) = \beta 0 + \beta 1 x$$

Equation 2 Least-Squares regression line

$$\beta 1 = r\frac{SDy}{SDx}$$

$$\beta 0 = \bar{Y} - \beta 1\,\bar{X}$$

$$r = \frac{n\sum_{i=1}^{n} x_i y_i - \{\sum_{i=1}^{n} x_i\}\{\sum_{i=1}^{n} y_i\}}{n\sum_{i=1}^{n} x_i^2 - \{\sum_{i=1}^{n} x_i\}}$$

Where,

- Y = the dependent variable
- $\beta 1$ = The slope of the regression line
- $\beta 0$ = The intercept point of the regression line and the y axis.
- $\bar{X}$ = Mean of x values
- $\bar{Y}$ = Mean of y values

- SD_x = Standard Deviation of x
- SD_y = Standard Deviation of y
- n = Number of observations

Using the sample data, β0 is equal to 26.57041699 and β1 is -0.044223446

The linear equation become

$$y = 26.57041699 - 0.044223446\,x$$

Figure 1 represents actual data in the blue line and the red line shows the linear function.

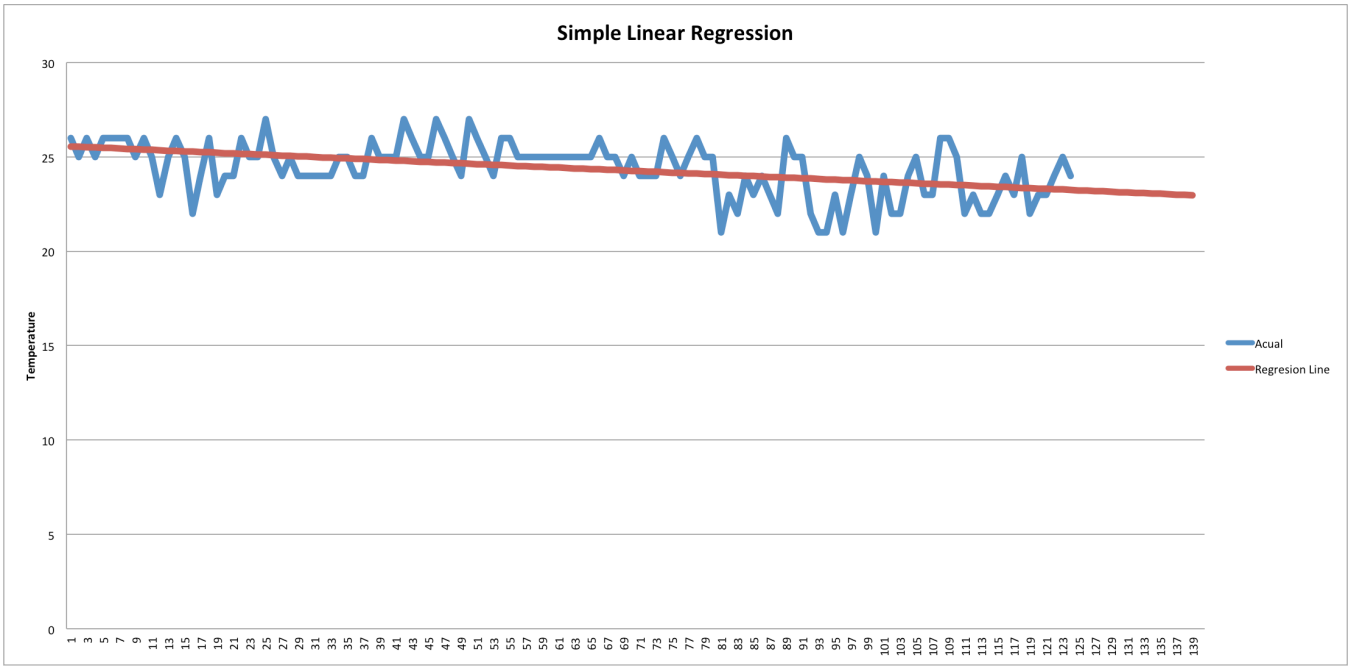

Figure 1 Reggresion line from obsereved data

As shown, future data could be predicted using the red line.

Using the mean square error formula it can be calculated the error is 1.376138377

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{Y}_i - Y_i\right)^2$$

Equation 3 Mean Square Error

3.4 Apache Spark Mlib for multiple linear regression

We typically have more than one informative variable; multiple linear regression models the relationship between two or more informative variables and a response variable by fitting a linear equation to observed data.

Underlining source code in calculates prediction for inside temperature having three more variables

- Inside humidity
- Outside temperature
- Outside humidity

Sample data are stored in alr.csv comma separated file formation like below

26,31.18614578,100,5.85
25,31.1844635,99.62,5.95
26,31.39333725,99.61,5.38
25,31.4606514,99.91,5.83
26,31.31670761,99.77,5.87
26,31.13899231,99.51,6.4

The first value of each line is inside temperature, second in inside humidity, the third variable is outside temperature and the last one is outside humidity

Underlying uses an SCV file and the linear regression method inside Spark to make predictions. The program's print calculates the mean square error by comparing forecasted data and real ones.

```
static <T> T toType(Class<T> aClass = Function, Closure closure) {
```

```
   [call: closure].asType(aClass)
}
def conf = new SparkConf().setMaster("local").setAppName("Linear Regression
Example")
def sc = new JavaSparkContext(conf)
def path = "data/lr.csv"
def data = sc.textFile(path)
def parsedData = data.map(toType { String line ->
   def parts = line.split(',').collect { it as double }
   new LabeledPoint(parts.find(), Vectors.dense(parts[1..3] as double[]))
})
parsedData.cache()
int numIterations = 9
def model = LinearRegressionWithSGD.train(JavaRDD.toRDD(parsedData),
numIterations)
def valuesAndPreds = parsedData.map(
       toType { LabeledPoint point ->
           double prediction = model.predict(point.features())
           new Tuple2(prediction, point.label())
       })
def rdd = valuesAndPreds.map(
       toType { Tuple2<Double, Double> pair ->
           Math.pow(pair._1() - pair._2(), 2.0)
       }).rdd()
def MSE = new JavaDoubleRDD(rdd)
println("Training Mean Squared Error =  ${MSE.mean()}")
```

The output for the program is

Training Mean Squared Error = 1.0159236909813228E70

CHAPTER 4

System design and implementation

A wireless sensor and building technology, created to develop a novel system for Self-learning Energy Efficient buildings and open Spaces (SEEDS), is used as an example of an IoT project with time-series data.

SEEDS is a research project supported by the 7th Framework Programme of the European Commission. It is a three-year project that started in September 2011.The project aims to develop an efficient energy management system that will allow buildings to continuously learn how to maintain user comfort while minimizing energy consumption and CO_2 emissions. SEEDS works with the development of ICT (Information and communications technology) tools to reduce energy use inside buildings and surrounding open spaces; this kind of tool is called Building Energy Management Systems (BEMS). The main goal of the tool is to optimize performance of the building (or group of buildings with their surrounding open spaces) in terms of comfort, functionality, energy efficiency, resource efficiency, economic return and lifecycle value.[19]

The time-series data are now residing in a relational DBMS (MySQL). The first goal is to migrate data into a scalable time-series database (OpenTSDB), developing generic query designs for the purpose of representing data in OpenTSDB. The second goal is to export data in OpenTSDB into a high-throughput distributed messaging system (Kafka), readable by a scalable machine learning library (Apache Spark) for the purpose of modeling in the machine learning environment, and in particular, for forecasting data.

Groovy, an object-oriented programming language for the Java platform, is used for implementation. The groovy code is dynamically compiled to Java Virtual Machine (JVM) byte-code, and interoperates with other Java code and libraries. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk.

Source code is available at https://github.com/maziarkaveh/OpenTSDB-seed/

4.1 Data structure in SEEDS

KE-HUS rooms have many wireless sensors mounted as part of the SEEDS project. Every minute, data values from wireless sensors are stored in appropriate tables in a MySQL database.

Each category has its own table for data insertion. For example, multiple sensors mounted on different floors of the building are responsible for sensing and reporting the temperature of the room to the **cipsi_seeds_uis_in_temp** table. Other tables for different categories of sensors include one for inside temperature, humidity, lighting, etc.

Since the nature of all data from various wireless sensors has the same structure, all tables use the same schema, as described below.

- **id:** an auto increment field for each record.
- **time:** the timestamp of when the event happens.
- **value:** the actual value of sensed data.
- **identity** and **idgui:** foreign keys to the wireless sensor device.

```
CREATE TABLE [Table name] (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `time` datetime NOT NULL,
  `value` double NOT NULL,
  `identity` int(11) DEFAULT NULL,
  `idgui` int(11) DEFAULT NULL,
```

```
  PRIMARY KEY (`id`),
  KEY `time` (`time`),
  KEY `identity` (`identity`),
  KEY `idgui` (`idgui`)
) ENGINE=InnoDB AUTO_INCREMENT=35973 DEFAULT CHARSET=utf8;
```

Figure 2 shows the raw data inside **cipsi_seeds_uis_in_temp** table

id	time (yyyy-MM-dd HH:mm:ss)	value	identity	idgui
2514894	2014-10-28 08:34:39	21.5499977112	264	718
2514895	2014-10-28 08:34:16	20.6800009155	374	828
2514896	2014-10-28 08:34:56	21.8199981689	269	723
2514897	2014-10-28 08:35:19	21.5699981689	264	718
2514898	2014-10-28 08:35:19	22.6799951172	314	768
2514899	2014-10-28 08:35:16	20.6899954224	374	828
2514900	2014-10-28 08:35:42	21.8099998474	269	723
2514901	2014-10-28 08:36:06	24.4299993896	364	818
2514902	2014-10-28 08:36:16	20.6599966431	374	828
2514903	2014-10-28 08:36:31	21.8300003052	269	723
2514904	2014-10-28 08:36:44	21.5699981689	264	718
2514905	2014-10-28 08:37:16	20.6800009155	374	828
2514906	2014-10-28 08:37:41	21.8199981689	269	723
2514907	2014-10-28 08:38:10	24.4199972534	364	818

*Figure 2 Row data inside **cipsi_seeds_uis_in_temp** MySQL table*

4.2 Migrate to OpenTSDB

The first phase of migration involves reading data from MySQL and exporting it to a time-series database.

Currently, the SEEDS database has 96 tables. As the nature of the data and table schema are similar, a generic class design to support all tables is created and a customized inherited class mad for each particular table.

Four tables are currently implemented. The framework supports more tables and

thus all tables could easily be implemented.

- **cipsi_seeds_uis_out_hum:** Recording humidity data outside the building
- **cipsi_seeds_uis_out_temp:** Recording temperature data outside the building
- **cipsi_seeds_uis_in_temp:** Recording temperature data inside the building
- **cipsi_seeds_uis_in_hum:** Recording humidity data inside the building

4.2.1 Domain Design

First, sensor identities need to be mapped to a static enum class to indicate the location of the sensor; this object will be used as tag metadata inside the OpenTSDB enum classes of LocationType and Room.

```
 enum LocationType {

   CONFERENCE_ROOM,OFFICE,CORRIDOR,CLASSROOM,AUDITORIUM,ROOF

}

enum Room {

   E_101(tempId: 27, humId: 28, room: 'KE-E101', floor: 1, locationType:
LocationType.AUDITORIUM),

   E_102(tempId: 33, humId: 34, room: 'KE-E102', floor: 1, locationType:
LocationType.AUDITORIUM),

   E_162(tempId: 39, humId: 40, room: 'KE-E162', floor: 1, locationType:
LocationType.CLASSROOM),

                                        .

                                        .

                                        .

   OUTDOOR(tempId: 460, humId: 461, room: 'ROOF', floor: 6, locationType:
LocationType.ROOF)

   Integer      tempId

   Integer      humId
```

```
    String       room
    Integer      floor
    LocationType locationType
}
```

The class **TSDBRecord** is going to be the parent class of all domain classes.

```
  abstract class TSDBRecord {
     Room   room
     Long   timestamp
     Number value
     abstract Room parsIdentity(Integer identity)
     abstract String getMetric()
     abstract String getTableName()
  }
```

Three abstract methods are available by which concrete table classes can be implemented.

- parsIdentity: takes the identity field in a table and finds the associated room object.
- getMetric: retrieves the name of the metric that is used in OpenTSDB.
- getTableName: retrieves the associated table name for fetching data from MySQL.

Note that the class contains other logic utility methods to be used in different places in the project. The complete source code is available on GitHub.

The source code for four concrete domain is:

```
class OutsideHumidityRecord extends TSDBRecord {
```

```
    final static String tableName = 'cipsi_seeds_uis_out_hum'
    final static String metric    = 'seeds.hum.out'
    @Override
    Room parsIdentity(Integer identity) {
        Room.findByHumidityIdentify(identity)
    }
}
class OutsideTemperatureRecord extends TSDBRecord {
    final static String tableName = 'cipsi_seeds_uis_out_temp'
    final static String metric    = 'seeds.temp.out'
    @Override
    Room parsIdentity(Integer identity) {
        Room.findByTemperatureIdentify(identity)
    }
}
class InsideTemperatureRecord extends TSDBRecord {
    final static String tableName = 'cipsi_seeds_uis_in_temp'
    final static String metric    = 'seeds.temp.in'
    @Override
    Room parsIdentity(Integer identity) {
        Room.findByTemperatureIdentify(identity)
    }
}
```

```
class InsideHumidityRecord extends TSDBRecord {

    final static String tableName = 'cipsi_seeds_uis_in_hum'

    final static String metric    = 'seeds.hum.in'

    @Override

    Room parsIdentity(Integer identity) {

        Room.findByHumidityIdentify(identity)

    }

}
```

The UML class diagram is shown in figure 3.

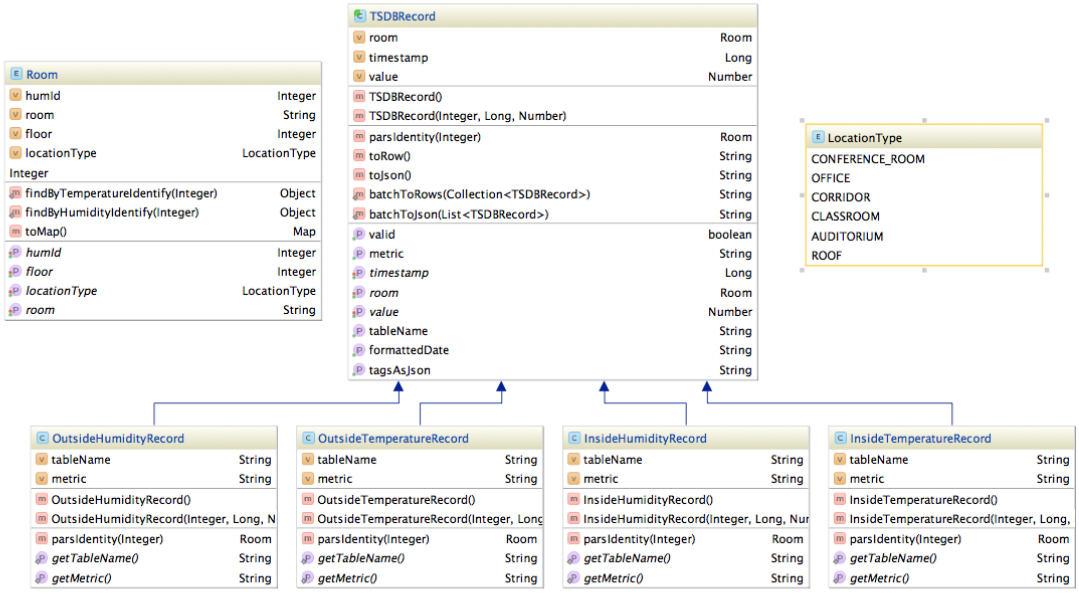

Figure 3 UML class diagram of domain classes of framework

4.2.2 Repository

A **RecordRepository** interface containing two methods is available.

```
 interface RecordRepository {

   public <T extends TSDBRecord> List<T> fetchAll(Class<T> type)

   void forEach(Class<? extends TSDBRecord> type, Closure closure)

}
```

The method **fetchAll** gets a subclass of records, taking TSDBRecord class type as parameter and returning a list of objects loaded from the database. The **forEach** method gets a Closure in addition to be run for every single record.

MysqlRecordRepository is an implementation class of the interface for MySql RDBMS.

4.2.3 Export data into OpenTSDB

As of now, three primary techniques to export data to OpenTSDB exist: Telnet API, HTTP API and file import from a file. HTTP API. Preparation batch files are implemented in the project as described below.

The import command in OpenTSDB does bulk loading of time-series data. One or more files are accepted and OpenTSDB will parse and load the data. Data must be organized in the Telnet put style with one data point for every line in a text file.

The **toRow** method in **TSDBRecord** class returns a Telnet style put string line.

```
 String toRow() {

   "$metric $timestamp $value room=${room.room} floor=${room.floor}
locationType=${room.locationType.name().toLowerCase()}"

}
```

The data in the import file should be arranged chronologically by timestamp. The **batchToRows** method takes a collection of TSDBRecord objects, filters out invalid data, sorts the records and then converts them to Telnet put format and joins them.

```
static String batchToRows(Collection<TSDBRecord> records) {

   def valid = records.findAll { it.valid }

   valid.sort { it.timestamp }.collect { it.toRow() }.join('\n')

}
```

The method **writeToFileForImport** in **OpenTSDBUtils** creates a file and writes formatted data into it.

```
 class OpenTSDBUtils {

   static writeToFileForImport(String path, Collection<TSDBRecord> data) {

       new File(path).text = TSDBRecord.batchToRows(data)

   }

}
```

This file can then be imported using CLI(**command-line** interface) in OpenTSDB.

The endpoint /api/put in OpenTSDB allows for moving data into OpenTSDB over HTTP as an alternative to the Telnet interface. **TimeSeriesDBHttpApi,** uses the **put** method to post a TSDBRecord object into OpenTSDB through HTTP API. The **query** object is used to query OpenTSDB records.

```
interface TimeSeriesDBHttpApi {

   void put(TSDBRecord tsdbRecord)

   def query(Query query)

}

class Query {

  Long                start
```

```
  Long                end
  String              aggregate
  String              metric
  Map<String, String> tags
}
```

CHAPTER 5

Kafka plugin for OpenTSDB

Streaming time-series data to Apache Spark requires pushing events into a cluster publish subscription framework. Currently Apache Kafka is one of the best choices. The next section provides background on Apache Kafka is, later there is a plugin developed for OpenTSDB to push new events happening in OpenTSDB into Kafka. That will be used later in Apache Spark as subscriber to process and analyzed the data.

5.1 What is Apache Kafka

In the present big-data period, the first challenge is to gather the information, as it is an immense amount of information. The second challenge is analyzing it. This analysis commonly incorporates the accompanying sorts of information, and considerably more:

- Client conduct information
- Application performance tracing
- Activity data as logs
- Event messages

Message distributed is a system for joining different applications with the assistance of messages that are directed between for instance, a message broker, for example, Kafka. Kafka is an answer for the real-time issues of any software solution; that is to say, dealing with continuous volumes of data and storing it to various consumers rapidly. Kafka consistently incorporates data from makers and customers without hindering the producers of the data and without telling producers

who the last consumers are.

Apache Kafka is an open source, distributed, partitioned, and replicated commit-log-based publish-subscribe messaging system, predominantly composed with the accompanying attributes:

- **Persistent messaging:** To get the genuine worth of enormous information, any sort of data loss cannot be managed. Apache Kafka is planned with O(1) disk structures that give constant-time performance even with vast volumes of persisting messages that are at the request of terabytes o dat. With Kafka, messages are stored on disk and replicated within the cluster to avoid information loss.
- **High throughput**: Keeping big data in mind, Kafka is intended to deal with commodity hardware and handles hundreds MBs of reads and writes every second from a massive number of customers.
- **Distributed**: Apache Kafka with its cluster-centric design explicitly supports message partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics. Kafka cluster can grow elastically and transparently without any downtime.
- **Multiple client support**: The Apache Kafka framework supports simple integration of clients from distinctive platforms, for example, Java, .NET, PHP, Ruby, and Python.
- **Real time**: Messages delivered by the producer threads ought to be quickly obvious to consumer threads; this element is discriminating to event-based systems, for example, Complex Event Processing (CEP) systems.

Kafka gives a real-time publish-subscribe solution that overcomes the challenges the real-time and batch data volumes that may develop all together of greatness to be bigger than the genuine information. Kafka additionally supports parallel information loading in the Hadoop frameworks.

The accompanying diagram demonstrates a commonplace big data aggregation-and-analysis situation upheld by the Apache Kafka messaging framework:

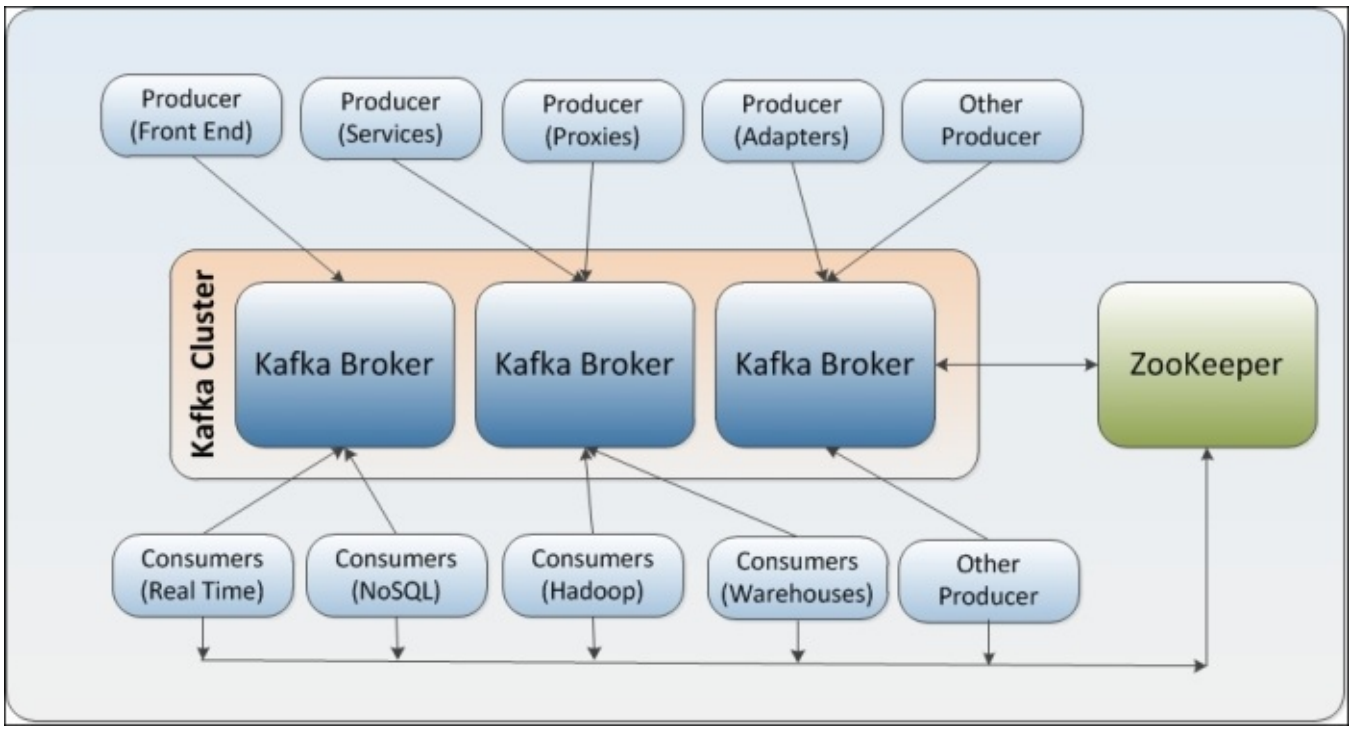

Figure 4 aggregate-and-analyze scenarios in Kafka messaging system [17]

On the production side, there are various types of producers, for example, the following:

- Frontend web applications generating application logs
- Producer proxies generating web analytics logs
- Producer adapters generating transformation logs
- Producer services, generating invocation trace logs[17]

On the consumption side, various types of consumers exist, for example:

- Offline consumers that are consuming messages and persisting them in Hadoop or conventional data warehouse for offline analysis.
- Near real-time consumers that consume messages and store them in any NoSQL data-store, for example, HBase or Cassandra, for near real-time analytics.
- Real-time consumers, for example, Spark or Storm, that filter messages in-memory and trigger alert events for related groups.

- Real-time usage of these numerous sets of information gathered from production frameworks has turned into a challenge due to the volume of information gathered and processed.

- Apache Kafka aims to bring together offline and online processing by providing a system to parallel load data in Hadoop frameworks and also the capacity to partition real-time utilization more than a cluster of machines. Kafka can be contrasted with Scribe or Flume as it is helpful for handling activity stream information; yet from the building design point of view, it is closer to legacy messaging frameworks, for example, ActiveMQ or RabitMQ.

Kafka can be utilized in the situation where gathered information, experiences preparing at numerous stages—an illustration is raw data devoured from topics and enhanced or changed into new Kafka topics for further consumption. Hence, such handling is likewise called stream processing.

Figure 5 demonstrates how publish-subscription messaging works inside Apache Kafka

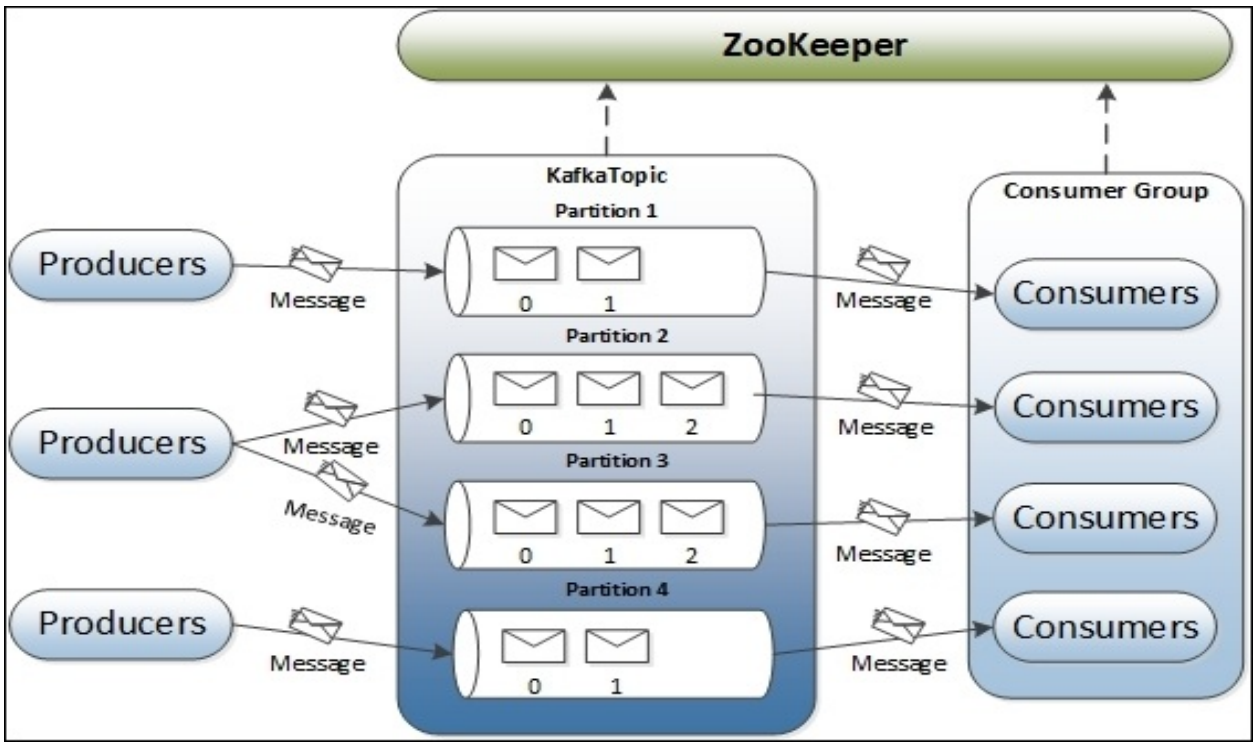

Figure 5 Messaging system in Kafka [17]

5.2 Kafka plugin for OpenTSDB

OpenTSDB 2.0 presents a plugin framework that will pushing data points received by a TSDB to be sent to Kafka in real time.

The implementation source code is available on GitHub:

https://github.com/maziarkaveh/OpenTSDBKafkaPublisher

5.2.1 Configure OpenTSDB

To enable real-time pushing, it is needed to set tsd.rtpublisher.enable config in OpenTSDB.config file and add tsd.rtpublisher.plugin property to the plugin name.

The following lines are used to enable Kafka plugin in OpenTSDB.

```
tsd.rtpublisher.enable = true
tsd.rtpublisher.plugin = no.uis.OpenTSDB.Kafkaplugin.KafkaPublisher
tsd.plugin.Kafka.broker.list = localhost:9092
tsd.plugin.Kafka.zookeeper.connect=localhost:2181
```

There is tsd.plugin.Kafka.broker.list property to the host and port of Kafka server and tsd.plugin.Kafka.zookeeper.connect for the zookeeper address.

5.2.2 Implementation

For real-time pushing plugin development one class should extend net.OpenTSDB.tsd.RTPublisher. KafkaPublisher is our plugin code and KafkaClient is responsible to push data into Kafka.

Figure 6 shows how the KafkaPublisher plugin works.

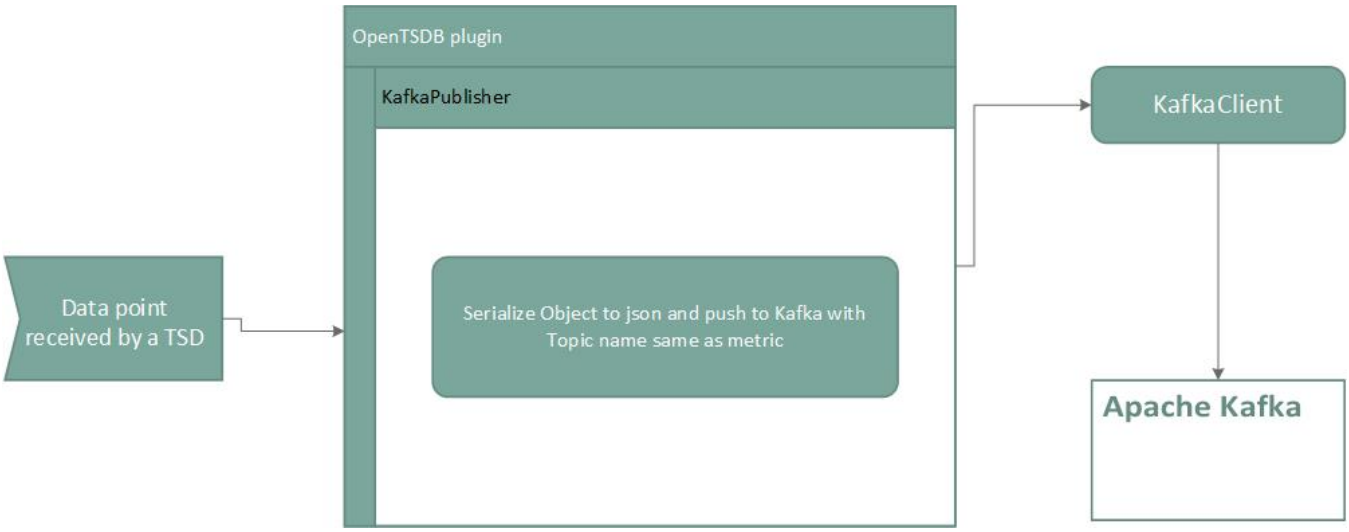

Figure 7 KakfaPublisher OpenTSDB plugin activity diagram

```
class KafkaPublisher extends RTPublisher {
   static final LOG = LoggerFactory.getLogger(KafkaPublisher)
   KafkaClient  KafkaClient
   void initialize(final TSDB tsdb) {
```

```
        def brokerList =
tsdb.config.getString('tsd.plugin.Kafka.broker.list')
        def zookeeperConnect =
tsdb.config.getString('tsd.plugin.Kafka.zookeeper.connect')
        if (!brokerList) {
            LOG.error('tsd.plugin.Kafka.broker.list is not set in
OpenTSDB.conf')
        }
        if (!zookeeperConnect) {
            LOG.error('tsd.plugin.Kafka.zookeeper.connect is not set in
OpenTSDB.conf')
        }
        KafkaClient = new KafkaClient(brokerList, zookeeperConnect)
        LOG.info('init initialize')
    }
    @Override
    Deferred<Object> publishAnnotation(Annotation annotation) {
        LOG.info('init publishAnnotation')
        new Deferred<Object>()
    }
    Deferred<Object> shutdown() {
        LOG.info('init shutdown')
        new Deferred<Object>()
    }
    String version() {
        LOG.info('init version')
        '0.1.0'
    }
    void collectStats(final StatsCollector collector) {
        LOG.info('init collectStats')
    }

    Deferred<Object> publishDataPoint(final String metric,
                                      final long timestamp, final long
value, final Map<String, String> tags,
```

```
                                        final byte[] tsuid) {
        publishDataPoint(metric, timestamp, value, tags, tsuid)
    }
    Deferred<Object> publishDataPoint(final String metric,
                                        final long timestamp, final double
value, final Map<String, String> tags,
                                        final byte[] tsuid) {
        LOG.info("publishDataPoint publishDataPoint $timestamp $metric
$value $tags $tsuid")
        KafkaClient.send([metric: metric, timestamp: timestamp, value:
value, tags: tags, tsuid: tsuid])
        new Deferred<Object>()
    }
  }

  class KafkaClient {
    Producer producer
    void send(Map data) {
        def msg = new KeyedMessage<String, String>(data.metric, 'spark',
JsonOutput.toJson(data))
        producer.send(msg)
    }
    KafkaClient(String brokerList, String zookeeperConnect) {
        this.producer = producer
        def props = new Properties()
        props.put('zookeeper.connect', zookeeperConnect);
        props.put('serializer.class', 'Kafka.serializer.StringEncoder')
        props.put('metadata.broker.list', brokerList);

        def config = new ProducerConfig(props)
        producer = new Producer<String, String>(config)
    }
```

5.2.3 Build and Deploy

The project is a maven project for compiling code and making a Jar file.

The build process is straight forward, all that it needed is to got to the project directory path and run:

```
mvn install

mvn assembly:single
```

Then the jar file KafkaPublisher-1.0-SNAPSHOT-jar-with-dependencies.jar is created in the target folder in project. The next step is to copy the jar file into the plugin folder in OpenTSDB and restart OpenTSDB.

As soon as there are new data inserted in OpenTSDB the data will be pushed into Kafka.

Now time-series data are available in Kafka with a same topic name as the metric name in OpenTSDB. Consumers (in our case Spark) can subscribe and stream data from Kafka.

CHAPTER 6

Test

The work done is divided into three levels. In the test environment, data are fetched from a MySQL database, wrapped in domain classes, and sent to OpenTSDB using restful API. Using a plug-in developed and deployed in OpenTSDB, time-series data are then pushed to Kafka in real-time. Finally, Spark, used as the streaming framework, listens for data in Kafka and sends data to Spark's engine for processing and to generate the desired results to the output. Figure 7 indicates the process.

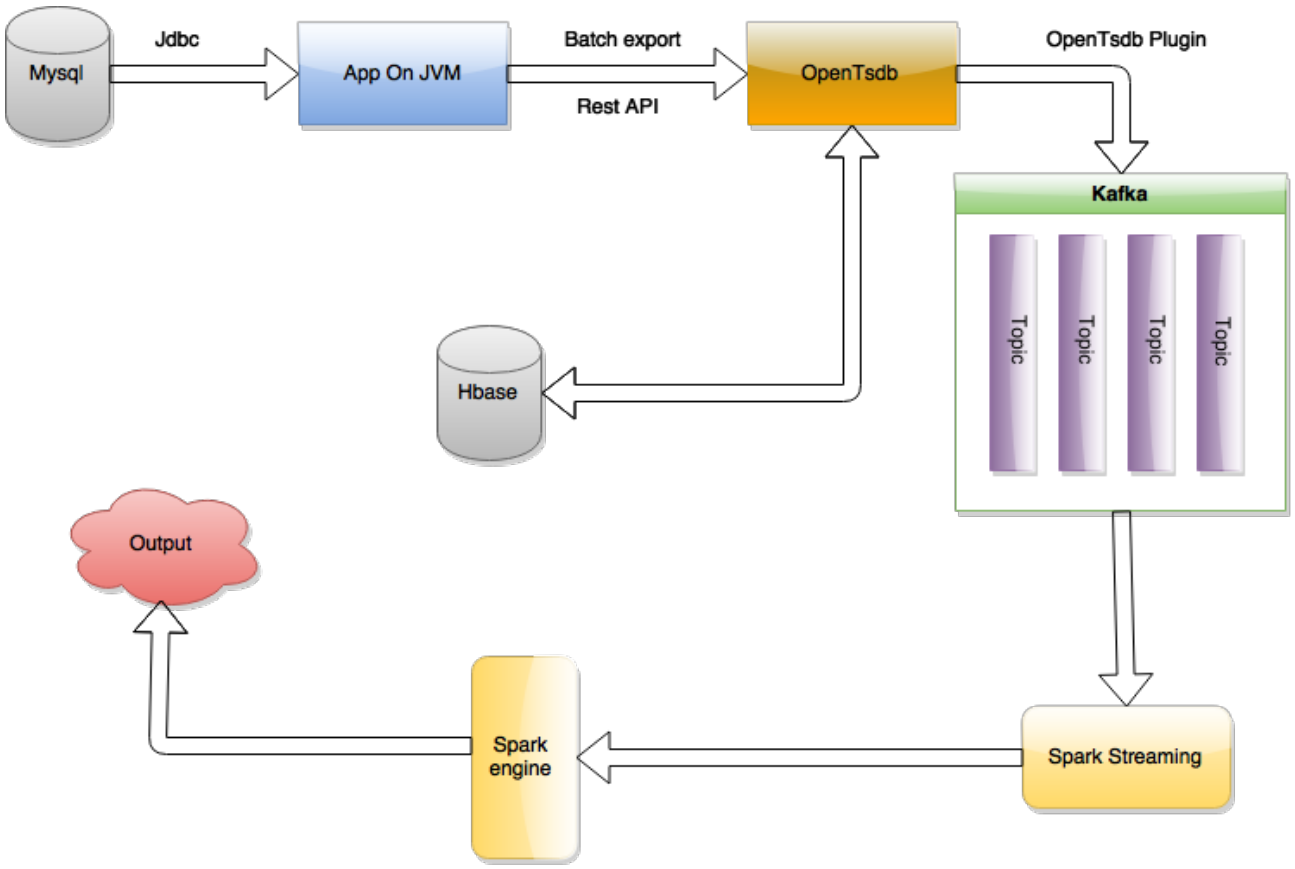

Figure 8 The complete framework process

6.1 simulation the framework

In order to simulate the real world environment, all four existing tables are loaded into RAM as a list of domain TSDBRecord objects.

We thus have four collections, each containing data for its associated table (OutsideHumidityRecord, OutsideTemperatureRecord, InsideHumidityRecord, InsideTemperatureRecord). Collections are grouped by time created and every second record is pushed to OpenTSDB. Thus, a multithread program is required with four threads to send the four types of data to OpenTSDB simultaneously.

The class PushTest is an implementation of the above-described simulation strategy. Figure 8 also shows activity diagram of the algorithm.

```
@Slf4j

class PushTest {

   static void main(String[] args) {

       GParsPool.withPool {

           def collect = [OutsideHumidityRecord, OutsideTemperatureRecord,
InsideHumidityRecord, InsideTemperatureRecord].collectParallel { record ->

               new MysqlRecordRepository(sql:
MySQLRecordRepository.DEFAULT_SQL).fetchAll(record).findAll {

                   it.valid

               }

           }.flatten().groupByParallel { it.timestamp }

           collect.each {

               it.value.eachParallel {

                   try {

                       OpenTSDBHttp.instance.put(it)

                       sleep 100

                   } catch (e) {

                       log.info(it.toJson())

                   }

               }

           }

       }

   }
```

}

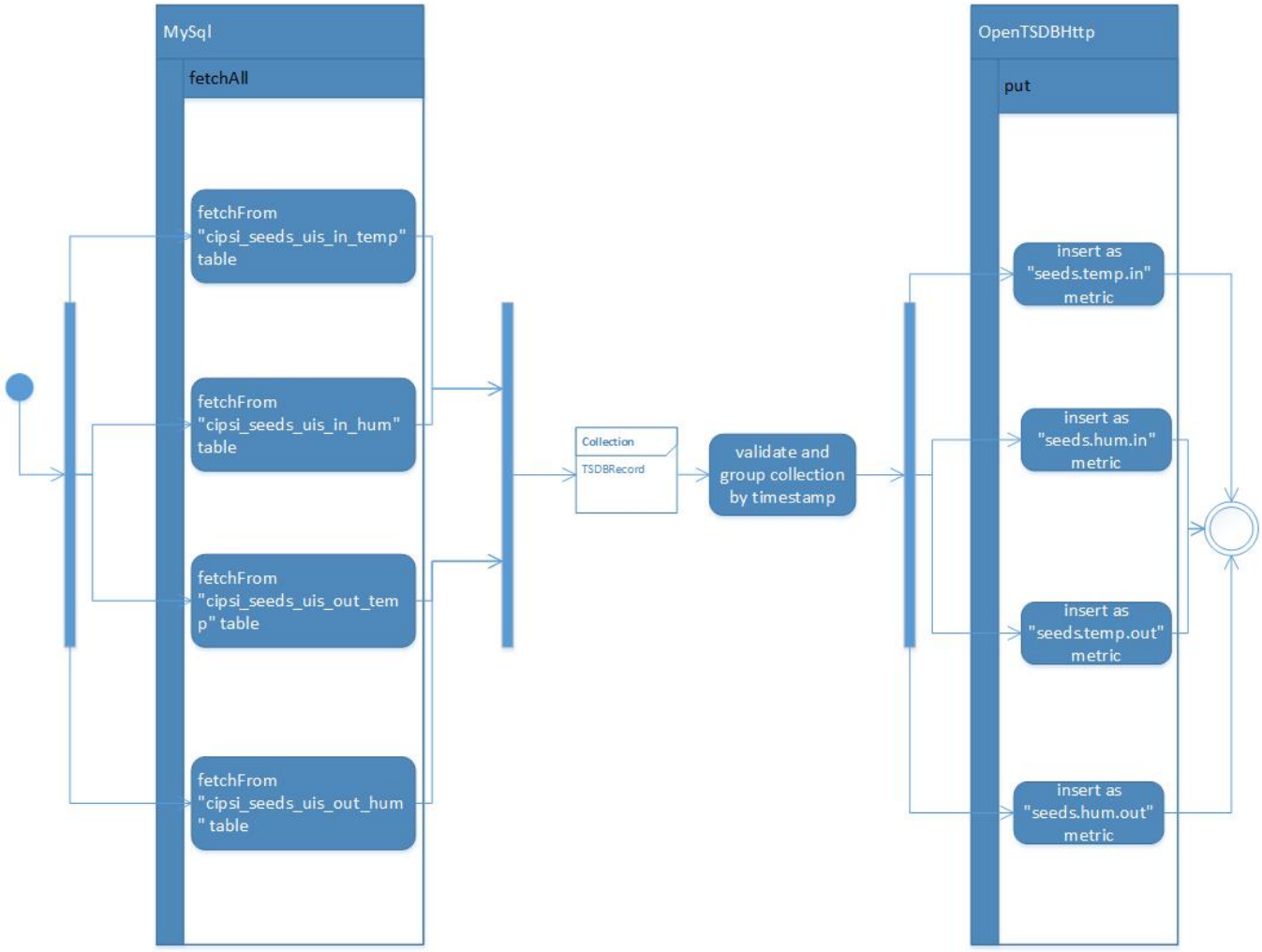

Figure 9 Test environment simulation activity diagram

The data are stored in an OpenTSDB database and also pushed into Kafka.

The next four figures show data inserted into OpenTSDB after running the PushTest program.

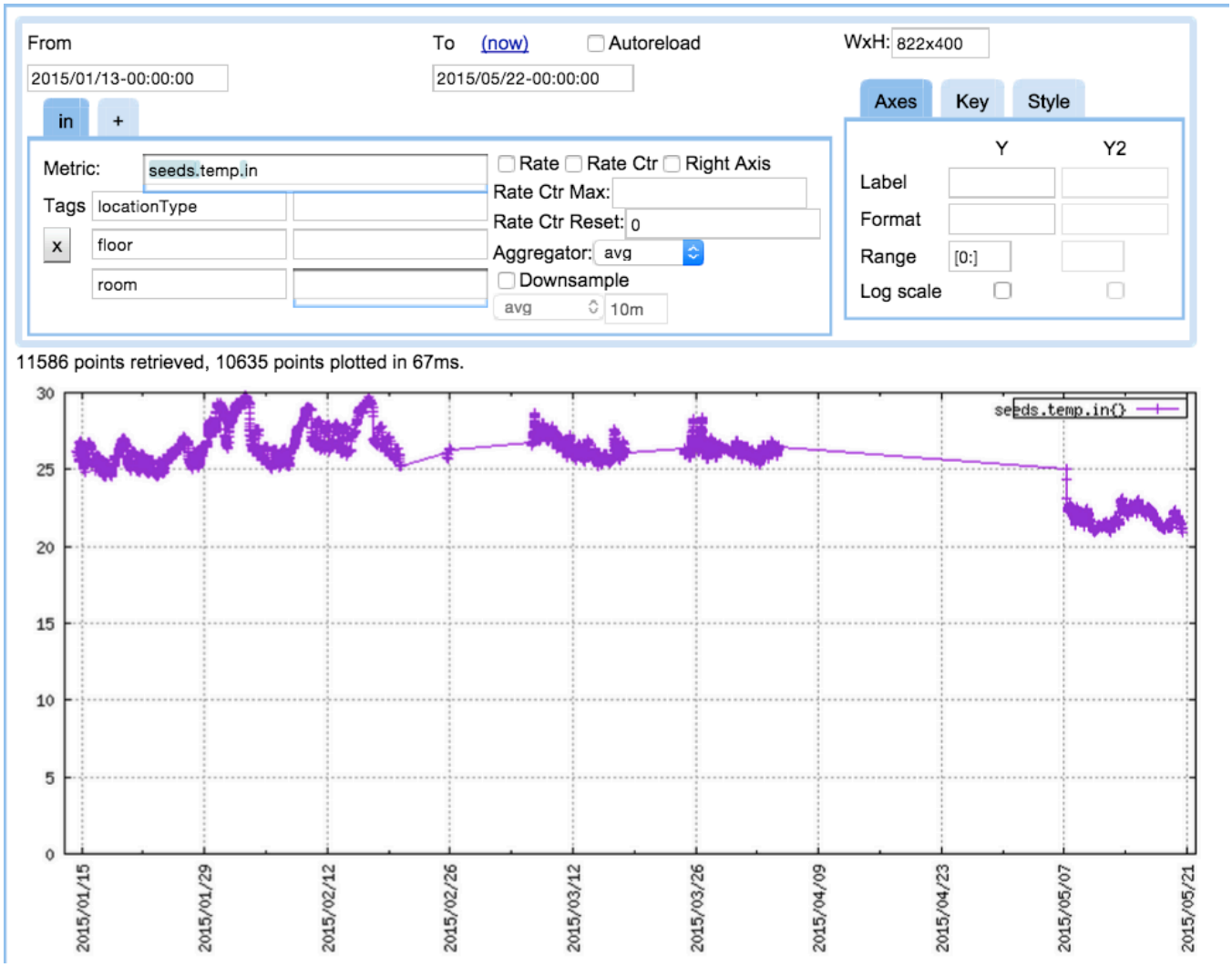

Figure 10 Inside temperature dada shown in OpenTSDB

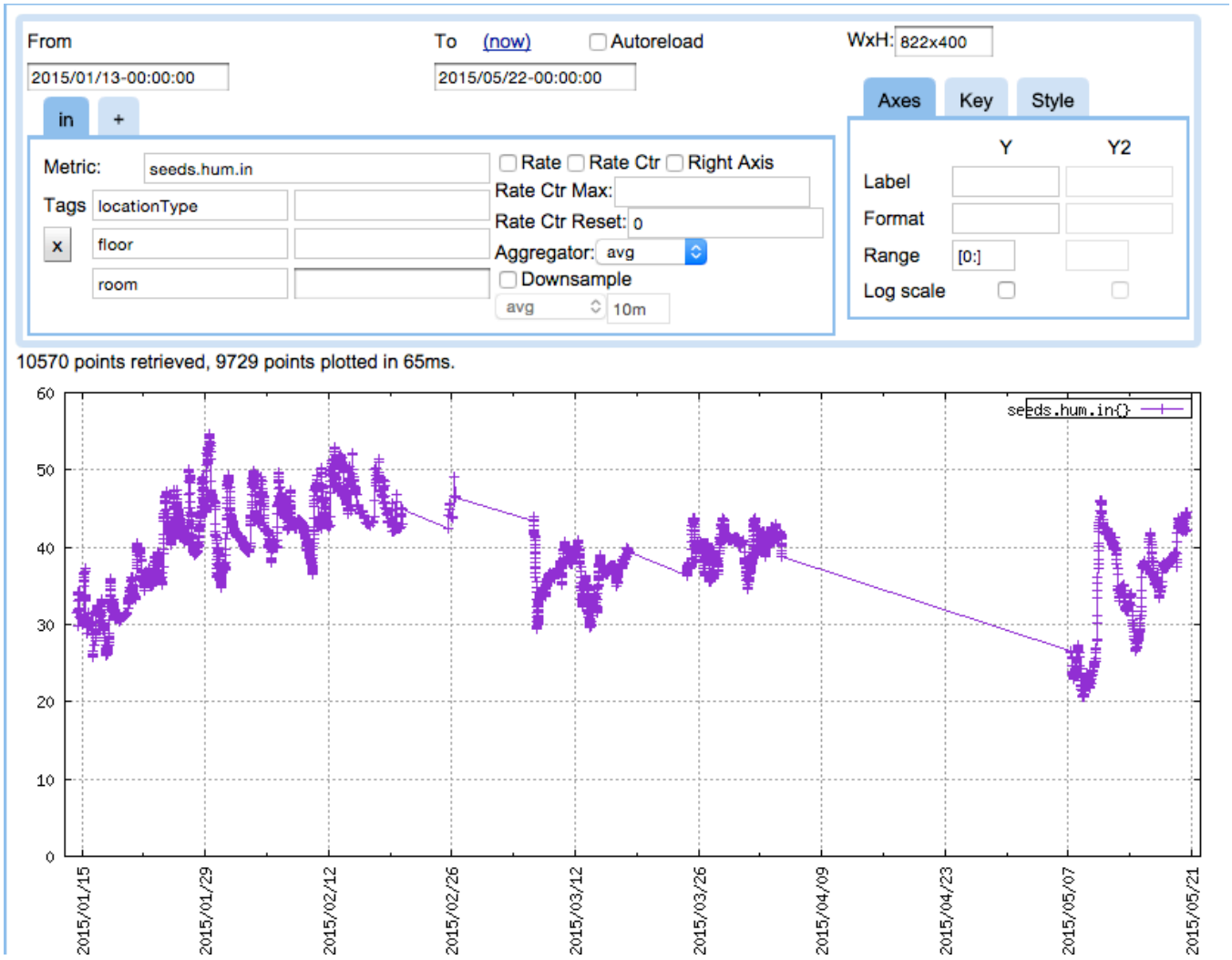

Figure 11 Inside humidity dada shown in OpenTSDB

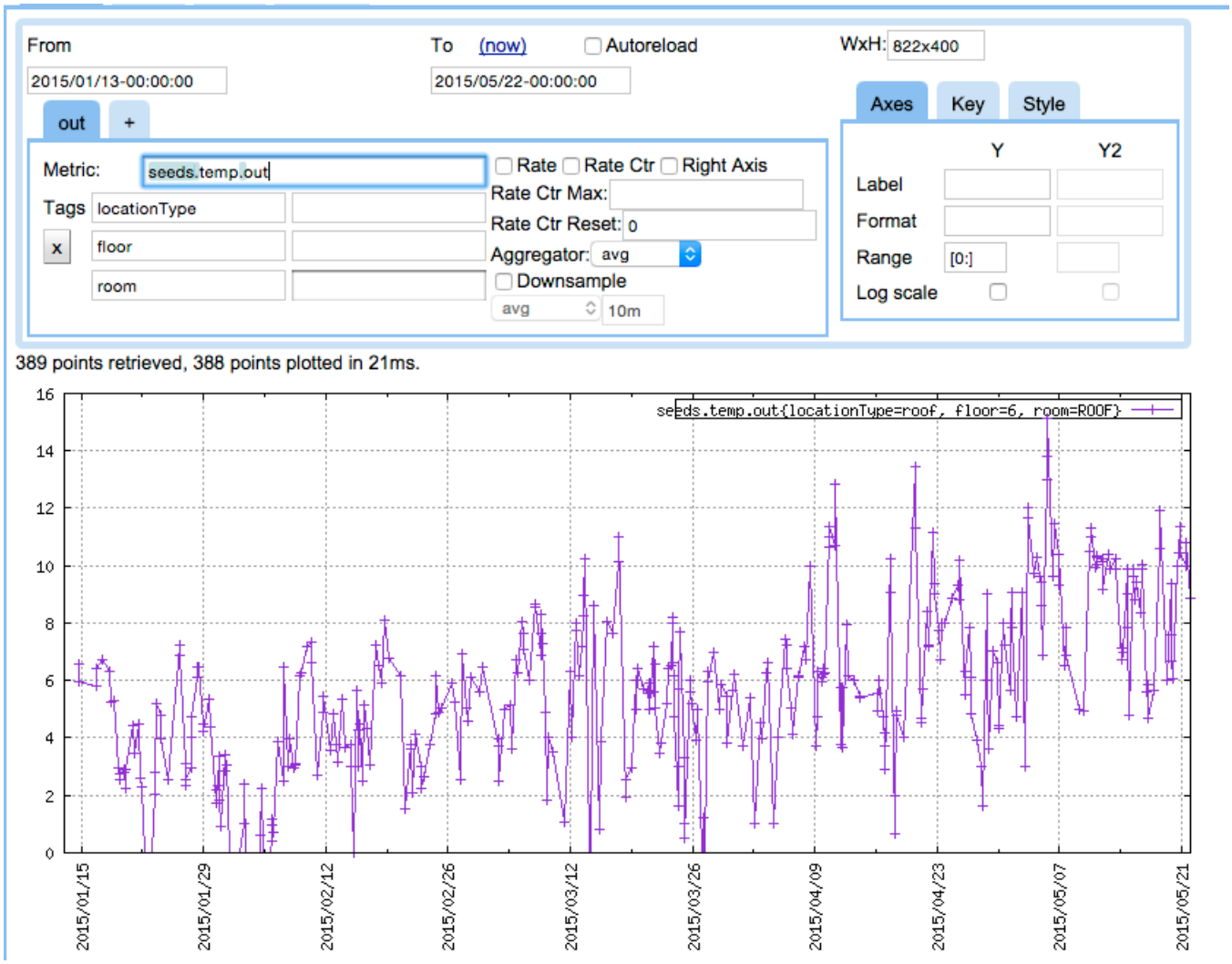

Figure 12 Outside temperature dada shown in OpenTSDB

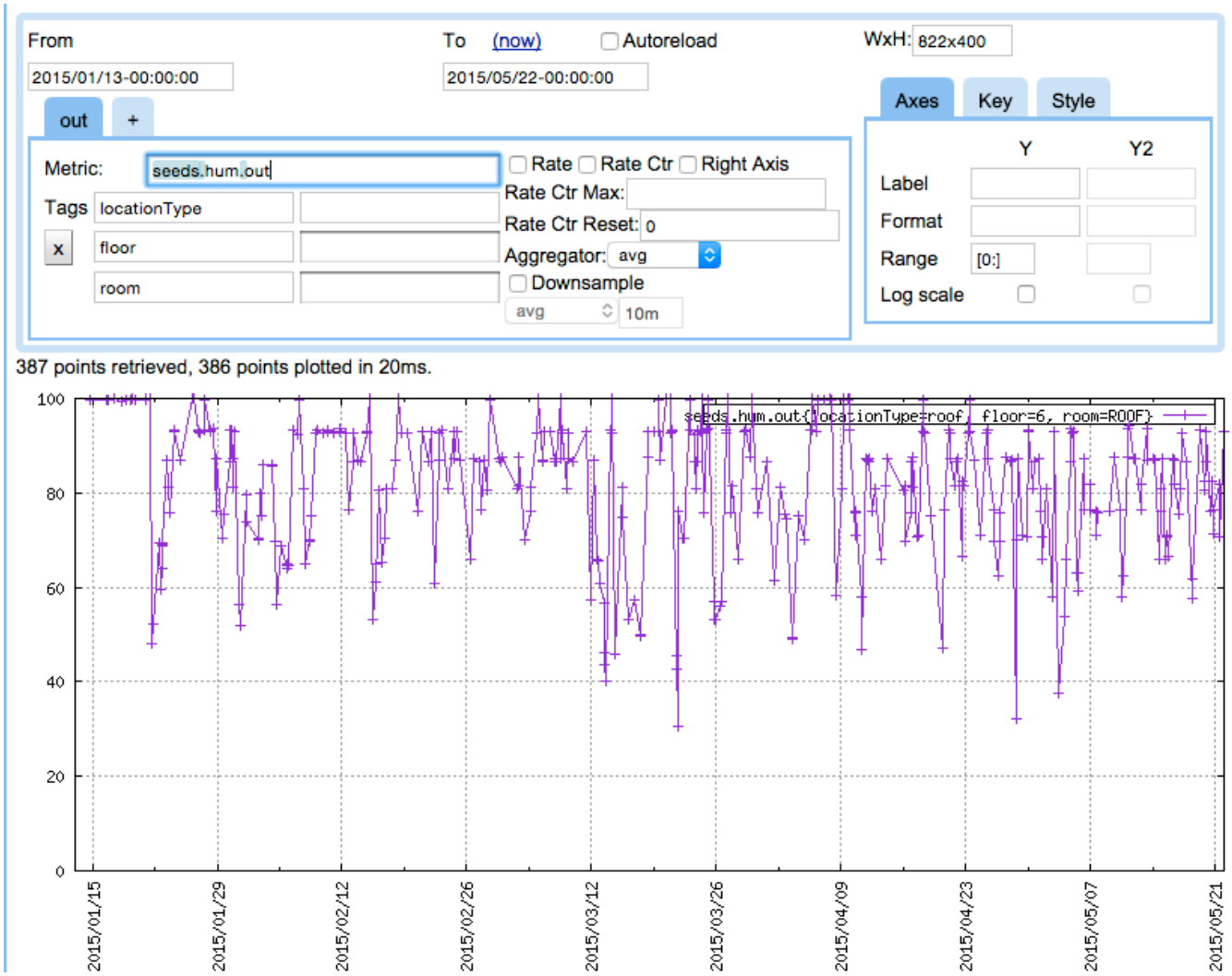

Figure 13 Outside humidity dada shown in OpenTSDB

JavaDirectKafka is the test class using Spark streaming to read data from Kafka every 10 seconds and de-serialize items into TSDBRecord objects. The objects are then available in Spark as four separate lists (inTemp, ouTemp, inHum, ouHum). The snippet only prints data to the output console, however as long time-series data are available in Spark's DStream instantly, they can be used for real-time analytics and machine learning in Spark.

```
class JavaDirectKafka {
```

```
    static <T> T toType(Class<T> aClass = Function, Closure closure) {

        [call: closure].asType(aClass)

    }

    static void main(String[] args) {

        String brokers = 'localhost:9092'

        def sparkConf = new
SparkConf().setMaster('local').setAppName('JavaDirectKafka')

        def jssc = new JavaStreamingContext(sparkConf, Durations.seconds(10))

        def topicsSet = [OutsideHumidityRecord, OutsideTemperatureRecord,
InsideHumidityRecord, InsideTemperatureRecord].collect {

            it.metric

        } as Set

        def KafkaParams = ['metadata.broker.list': brokers]

        def messages = KafkaUtils.createDirectStream(

                jssc, String, String, StringDecoder, StringDecoder,
KafkaParams, topicsSet

        )

        JavaDStream<? extends TSDBRecord> records = messages.map(toType {
tuple2 ->

            TSDBRecord.fromJson(tuple2._2())

        })

        def inTemp = records.filter(toType { TSDBRecord record ->
record.metric == InsideTemperatureRecord.metric })

        def ouTemp = records.filter(toType { TSDBRecord record ->
record.metric == OutsideTemperatureRecord.metric })
```

```
        def inHum = records.filter(toType { TSDBRecord record ->
record.metric == InsideHumidityRecord.metric })

        def ouHum = records.filter(toType { TSDBRecord record ->
record.metric == OutsideHumidityRecord.metric })



        inTemp.print()

        ouTemp.print()

        inHum.print()

        ouHum.print()

        jssc.start()

        jssc.awaitTermination()

    }

}
```

Here is the out of the program in console

Time: 1433944560000 ms

InsideTemperatureRecord(super:TSDBRecord(room:E_101, timestamp:1422685913000, value:26.979995727539062))

InsideTemperatureRecord(super:TSDBRecord(room:E_423A, timestamp:1425831689000, value:28.019996643066406))

InsideTemperatureRecord(super:TSDBRecord(room:E_427, timestamp:1427638347000, value:25.209999084472656))

InsideTemperatureRecord(super:TSDBRecord(room:E_101,

timestamp:1423535272000, value:26.93999481201172))

InsideTemperatureRecord(super:TSDBRecord(room:E_164,
timestamp:1423580362000, value:26.089996337890625))

InsideTemperatureRecord(super:TSDBRecord(room:E_166,
timestamp:1423580362000, value:27.029998779296875))

InsideTemperatureRecord(super:TSDBRecord(room:E_429,
timestamp:1425820154000, value:27.860000610351562))

InsideTemperatureRecord(super:TSDBRecord(room:E_470,
timestamp:1427752644000, value:29.029998779296875))

InsideTemperatureRecord(super:TSDBRecord(room:E_462,
timestamp:1431993084000, value:19.93999481201172))

InsideTemperatureRecord(super:TSDBRecord(room:E_425B,
timestamp:1426093771000, value:25.979995727539062))

...

Time: 1433944560000 ms

OutsideTemperatureRecord(super:TSDBRecord(room:OUTDOOR,
timestamp:1431754005000, value:8.369999885559082))

Time: 1433944560000 ms

InsideHumidityRecord(super:TSDBRecord(room:E_101,
timestamp:1422685913000, value:39.35163116455078))

InsideHumidityRecord(super:TSDBRecord(room:E_423A,
timestamp:1425831689000, value:32.423606872558594))

InsideHumidityRecord(super:TSDBRecord(room:E_427,
timestamp:1427638347000, value:43.01470184326172))

InsideHumidityRecord(super:TSDBRecord(room:E_101,
timestamp:1423535272000, value:36.52838897705078))

InsideHumidityRecord(super:TSDBRecord(room:E_166,
timestamp:1423580362000, value:46.34523391723633))

InsideHumidityRecord(super:TSDBRecord(room:E_164,
timestamp:1423580362000, value:48.67914962768555))

InsideHumidityRecord(super:TSDBRecord(room:E_429,
timestamp:1425820154000, value:35.36179733276367))

InsideHumidityRecord(super:TSDBRecord(room:E_470,
timestamp:1427752644000, value:37.0054931640625))

InsideHumidityRecord(super:TSDBRecord(room:E_462,
timestamp:1431993084000, value:40.74102020263672))

InsideHumidityRecord(super:TSDBRecord(room:E_425B,
timestamp:1426093771000, value:39.82991409301758))

...

Time: 1433944560000 ms

OutsideHumidityRecord(super:TSDBRecord(room:OUTDOOR,
timestamp:1431754005000, value:87.37000274658203))

CHAPTER 7

Conclusion

In this thesis, I introduce IoT and demonstrate how data can be assigned to a time-series database. Different available open-source time-series databases are introduced and compared, and OpenTSDB selected for implementation of the proposed method.

A platform is created to migrate time-series data in legacy systems from relational databases to OpenTSDB. The code is easily expandable for any kind of data in any database, as long as data are time-series data. The data are then viewed using the OpenTSDB graph view.

In order to process real-time, time-series data from OpenTSDB using analytical tools (Apache Spark), a plug-in is developed to export data from OpenTSDB real-time to Apache Kafka.

In the last section, I simulate an example of real data coming from MySQL and then view the available data in Spark real-time.

7.1 Limitation and future work

All frameworks developed in source code, including Zookeeper, HBase, OpenTSDB, Kafka and Spark, are hard-coded to run on the local computer. However, in real-time implementation they can be run in cluster with multiple machines by preparing the cluster environment appropriately.

While time-series data are available, real-time in Spark using Spark streaming, future work on analysis of the data is possible, using Spark Mlib for machine learning and Spark GraphX to display analyzed data.

Reference

[1] Apache HBase- http://hbase.apache.org

[2] Cassandra - http://cassandra.apache.org/http://cassandra.apache.org/

[3] Eben Hewitt - Cassandra: The Definitive Guide

[4] John A. Stankovic - Real-Time Systems Series

[5] OpenTSDB- http://OpenTSDB.net/.

[6] Kairosdb - http://kairosdb.github.io/kairosdocs/index.html.

[7] Druid - http://druid.io/docs/0.6.160/http://druid.io/docs/0.6.160/

[8] Influxdb - http://influxdb.com/http://influxdb.com/

[9] TSDB - http://en.wikipedia.org/wiki/Time_series_database.

[10] Ted Dunning , Ellen Friedman - Time Series Databases New Ways to Store and Access Data

[11] John A. Stankovic - Real-Time Systems Series Series

[12] OpenTSDB - A Distributed, Scalable Monitoring System net

[13] Apache Spark - https://spark.apache.org/

[14] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia ,Learning Spark Lightning-Fast Big Data Analysis

[15] Nick Pentreath -Machine Learning with Spark Progressing

[16] Apache Kafka - https://Kafka.apache.org/

[17] Nishant Garg - Learning Apache Kafka, 2nd Edition

[18] Jay Kreps,Neha Narkhede,Jun Rao - Kafka: a Distributed Messaging System for Log Processing

[19] SEEDS project - http://seeds-fp7.eu/index2.php

[20] Francis daCosta -Rethinking the Internet of Things

[21] Thomas Goldschmidt, Anton Jansen, Heiko Koziolek, Jens Doppelhamer, Hongyu Pei Breivold - Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes

[22] Ankur Khetrapal, Vinay Ganesh - HBase and Hypertable for large scale distributed storage systems

[23] Fangjin Yang , - Druid, A Real-time Analytical Data Store

[24] T. W. Wlodarczyk - Overview of time series storage and processing in a cloud environment

[25] - Tomasz Wiktor Wlodarczyk - Performance of Data Extraction from OpenTSDB