

University of Bern

Master's Thesis

---

# Real-Time Anomaly Detection in Water Distribution Networks using Spark Streaming

---

Stefan Nüesch

from Wabern

## Supervisors

Prof. Dr. Philippe Cudré-Mauroux  
Djellel Eddine Difallah

## Research Group

eXascale Infolab (Xi)  
Department of Informatics  
University of Fribourg

November 2014

# Abstract

This thesis analyses the adequacy and the performance of the Apache Spark Streaming engine in the context of anomaly detection in water distribution networks (WDN). It builds on an already proposed scenario in which sensors are extensively deployed in a WDN allowing for distributed, near-real-time anomaly detection. For this purpose, it uses several variations of LISA statistics and according statistical tests.

In order to show that *Spark Streaming* is applicable to such a setting, algorithms computing these statistics in *Spark Streaming* were developed. Subsequently, the resulting prototype was tested in a simulated WDN setting. Finally, the performance of the different algorithms as well as the impact of several network characteristics were measured.

The results show that the calculation of LISA statistics can be achieved with reasonable performance by using *Spark Streaming*. Furthermore, they reveal certain characteristics and limitations of *Spark Streaming*.

# Acknowledgements

First and foremost, I would like to thank Djellel Eddine Difallah and Professor Philippe Cudré-Mauroux who supervised my work for their great support throughout this thesis. Without their advice and ideas, this thesis would not have been possible.

I would also like to express my gratitude to the staff at eXascale Infolab<sup>1</sup> for providing me access to their cluster and for supporting me in resolving issues concerning this cluster.

I would especially like to thank Bettina Stähli and Michael Hofer for their support and their effort to correct this thesis. Their constructive criticism helped me to greatly improve this thesis.

---

<sup>1</sup><http://www.exascale.info>

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>1</b>  |
| <b>2</b> | <b>Related Work</b>   | <b>3</b>  |
| 2.1      | Local Indicators of Spatial Association . . . . .           | 3         |
| 2.1.1    | Spatial LISA . . . . .                                      | 4         |
| 2.1.2    | Cluster / Outlier Type . . . . .                            | 4         |
| 2.1.3    | Temporal LISA . . . . .                                     | 5         |
| 2.1.4    | Statistical Significance - Monte Carlo Simulation . . . . . | 5         |
| 2.2      | <i>Apache Spark</i> and <i>Spark Streaming</i> . . . . .    | 7         |
| 2.2.1    | Spark Architecture . . . . .                                | 7         |
| 2.2.2    | Resilient Distributed Datasets . . . . .                    | 8         |
| 2.2.3    | Dependencies and Scheduling . . . . .                       | 9         |
| 2.2.4    | <i>Spark Streaming</i> . . . . .                            | 10        |
| 2.2.4.1  | Discretised Streams . . . . .                               | 11        |
| 2.2.4.2  | Data Input . . . . .  | 12        |
| 2.2.5    | Spark Streaming vs. Apache Storm . . . . .                  | 12        |
| 2.3      | Anomaly Detection in Water Distribution Networks . . . . .  | 13        |
| <b>3</b> | <b>Implementation</b>                                       | <b>15</b> |
| 3.1      | Architecture . . . . .                                      | 15        |
| 3.1.1    | Network Topology . . . . .                                  | 16        |
| 3.1.2    | Receiver . . . . .  | 18        |
| 3.1.3    | Output Processing . . . . .                                 | 19        |
| 3.2      | Algorithms . . . . .  | 19        |
| 3.2.1    | Spatial LISA . . . . .                                      | 20        |
| 3.2.2    | LISA with Temporal Association . . . . .                    | 22        |
| 3.2.3    | Spatial Monte Carlo Simulation . . . . .                    | 23        |
| 3.2.4    | Temporal Monte Carlo Simulation . . . . .                   | 25        |

|          |   |           |
|----------|---|-----------|
| <b>4</b> | <b>Experiments</b>  | <b>28</b> |
| 4.1      | Setting . . . . .   | 28        |
| 4.1.1    | Test Bed . . . . .  | 28        |
| 4.1.2    | Topology . . . . .  | 29        |
| 4.1.3    | Input / Output . . . . .  | 29        |
| 4.2      | Spatial LISA Calculation . . . . .                                  | 29        |
| 4.3      | LISA with Temporal Association . . . . .                            | 34        |
| 4.4      | Impact of Topology Density . . . . .                                | 35        |
| 4.5      | LISA with Monte Carlo Simulation . . . . .                          | 37        |
| 4.6      | LISA with Temporal Association and Monte Carlo Simulation . . . . . | 40        |
| <b>5</b> | <b>Conclusion</b>   | <b>42</b> |
| 5.1      | Future Work . . . . .   | 43        |
| <b>A</b> | <b>Appendix</b>   | <b>48</b> |
| A.1      | Complete Source Code . . . . .                                      | 48        |

# List of Figures

|      |   |    |
|------|---|----|
| 2.1  | Spark Program Architecture (source: [21]) . . . . .                 | 8  |
| 2.2  | Spark Dependencies (source: [21]) . . . . .                         | 9  |
| 2.3  | Spark Scheduling Stages (source: [21]) . . . . .                    | 10 |
| 2.4  | DStream Processing Model (source: [23]) . . . . .                   | 11 |
| 3.1  | Application Architecture Overview . . . . .                         | 16 |
| 3.2  | Simple Topology Model (adapted from [18]) . . . . .                 | 17 |
| 3.3  | Legend . . . . .  | 20 |
| 3.4  | Spatial LISA Algorithm . . . . .                                    | 21 |
| 3.5  | Temporal LISA Algorithm . . . . .                                   | 22 |
| 3.6  | Spatial Monte Carlo Simulation - Naive Approach . . . . .           | 23 |
| 3.7  | Spatial Monte Carlo Simulation - Improved Approach . . . . .        | 24 |
| 3.8  | Temporal Monte Carlo Simulation . . . . .                           | 25 |
| 4.1  | Single LISA Run for 1 and 16 Nodes . . . . .                        | 30 |
| 4.2  | Average Calculation Times . . . . .                                 | 31 |
| 4.3  | Sample CPU Usage in the Cluster . . . . .                           | 33 |
| 4.4  | Sample Network Usage in the Cluster . . . . .                       | 33 |
| 4.5  | Temporal LISA - Average Duration for Different K's . . . . .        | 35 |
| 4.6  | Average Calculation Duration for Different Topology Types . . . . . | 36 |
| 4.7  | Scheduling Delays - Naive Approach . . . . .                        | 37 |
| 4.8  | Monte Carlo Simulation Performance - Naive Approach . . . . .       | 38 |
| 4.9  | Improved Monte Carlo Simulation Performance . . . . .               | 39 |
| 4.10 | Temporal Monte Carlo - Algorithm Comparison . . . . .               | 40 |

# List of Tables

|     |  |    |
|-----|--|----|
| 4.1 | Cluster Node Specifications (adapted from [20]) . . . . .          | 29 |
| 4.2 | HDFS Save Duration per Number of Workers . . . . .                 | 31 |
| 4.3 | Average Calculation Time per Number of Nodes and Cluster Workers . | 32 |
| 4.4 | Calculation Durations for Topology Types . . . . .                 | 36 |
| 4.5 | Average Calculation Duration in Seconds . . . . .                  | 39 |

# List of listings

|   |  |    |
|---|--|----|
| 1 | Sample Proximity Matrix File . . . . .                     | 18 |
| 2 | Global Selection of Random Past Neighbour Values . . . . . | 26 |
| 3 | Local Selection of Random Past Neighbour Values . . . . .  | 26 |



# List of Acronyms

**ADBMS** Array Database Management System.

**API** Application Programming Interface.

**CSR** Complete Spatial Randomness.

**DStream** Discretised Stream.

**HDFS** Hadoop File System.

**JVM** Java Virtual Machine.

**LISA** Local Indicators of Spacial Association.

**RDD** Resilient Distributed Dataset.

**WDN** Water Distribution Network.

**YARN** Yet Another Resource Negotiator (Hadoop MapReduce 2.0).

# 1

## Introduction

The ever growing availability of communication technology has lead to vast amounts of data being generated from the electronic monitoring of city infrastructure in today's urban areas. With emerging technologies capable of processing large amounts of data, efforts to leverage its value and to improve existing infrastructure have increased. The research field pursuing these efforts is called Smarter Cities, and it covers numerous types of infrastructure including public and private transport, electrical networks and communication infrastructure.

The present thesis deals with a particular type of infrastructure, namely water distribution. In traditional water distribution networks (WDNs), anomaly detection relies heavily on human labour, e.g. manual water quality sampling and reports by citizens. Automated monitoring usually only includes hydraulic parameters such as water pressure and flow, and covers only small parts of the WDN. Consequently, anomaly detection suffers from high delays in traditional WDNs. Reacting to time sensitive issues such as theft and leakage is therefore difficult [13].

D.E. Difallah, P. Cudré-Mauroux, and S.A McKenna proposed the use of Smarter Cities technologies as potential solution to these issues [13]. Their scenario envisages to deploy affordable sensors throughout the entire WDN which measure different characteristics, such as pressure, flow or water quality. These sensors send their measurements wirelessly to intermediate computing stations (called base stations). Ultimately, the sensor data is

stored in an Array Database Management System (ADBMS). The goal of this implementation is to detect anomalies in the network by using particular statistical metrics, which are calculated on the base stations and stored along with the sensor data in the ADBMS. It has been shown that the architecture and the methods used in this approach are feasible to detect anomalies in large scale WDN. Ongoing research in several related fields, however, has opened opportunities for improvements. In particular, stream processing systems combinable with Big Data technologies have become available and are constantly improving. This thesis evaluates such a system in the context of the WDN outlined above.

In detail, the goal of this thesis is to implement the statistical calculations proposed in [13], and to evaluate the performance of *Spark Streaming* in this scenario. For this purpose, a prototype application is built to fit the architecture of the proposed system whose performance is then evaluated in various settings.

# 2

## Related Work

As mentioned in the previous chapter, the goal of this thesis is to implement statistical calculations with *Spark Streaming* and to evaluate the performance of the solution. In the following chapter, the theoretical background necessary for this purpose is explained in three parts. In detail, the first part covers the statistical foundation for the algorithm implementation. In the second part, the Big Data platform used for the implementation is explained. The third part briefly summarises the WDN application proposed in [13].

### 2.1 Local Indicators of Spatial Association

Spatial correlation statistics have been a part of geographical studies for many years. They provide a means for detecting clusters of similar values and outliers. Local indicators of spatial association (LISA) were first introduced for cluster and outlier detection in geographical studies. They are based on observations of the same property at different points in a two-dimensional space and express relationships between such observations weighted by the spatial distance between them [12].

Research has shown that LISA statistics may also be used as a means to detect local anomalies in WDNs [13]. For this purpose, they modelled the network topology as a proximity-matrix which contains, for each sensor, a row of distance values. These values

represent the length of the water pipe connection between two sensors (with length=0 if two sensors are not connected).

For this thesis, a simplified version of this model is used, in which to each connection between two nodes a weight of 1 is assigned. Nodes which are not directly interconnected in the network are treated as non-related. Furthermore, in order to apply the originally time stationary LISA indicators to a constant stream of sensor observations, the indicators are repeatedly calculated for observations recorded in distinct time intervals.

### 2.1.1 Spatial LISA

The spatial, time stationary LISA statistics used throughout this thesis is the local Moran's  $I$ . It is defined in Equation 2.1, where  $v_a$  is the observed value at position  $a$ ,  $N$  is the number of neighbouring (i.e. connected) sensors,  $v_n$  is the measured value at neighbour  $n$ ,  $m$  is the mean of all measured values and  $S$  is their standard deviation [14].

$$LISA(v_a) = \left( \frac{v_a - m}{S} \right) \left[ \sum_{n=1}^N \left( \frac{1}{N} \right) \left( \frac{v_n - m}{S} \right) \right]$$

Equation 2.1: Spatial LISA

Spatial outliers are represented by negative values for  $LISA(v_a)$ , while positive values indicate clusters. Furthermore, the "magnitude [of the value] informs on the extent to which [original measurements] and neighbourhood values differ" [14].

### 2.1.2 Cluster / Outlier Type

While the local Moran's  $I$  statistics provide a means to detect clusters and outliers, they are not feasible to determine the cluster type. A strongly positive  $LISA(v_a) = I$ , for example, identifies a local cluster. However, it cannot be used to determine if the cluster is a so-called hot-spot (high value amidst other high values, also called high-high or HH) or a cold-spot (low value amidst low values, also called low-low or LL). The same applies to outliers (LH / HL) identified by negative values for  $I$ . While methods to determine types of clusters and outliers have been proposed (e.g. Getis-Ord  $G_i$  and  $G_i^*$  [12]), they are not within the scope of this thesis [12] [14].

### 2.1.3 Temporal LISA

In order to apply LISA to WDNs, an extension to the spatial Moran's  $I$  which includes a temporal dimension was proposed in [13]. More specifically, "K [was enlarged] around a node  $a$  to include its own previous measurements in addition to both the current and past measurements from its neighbours"[13]. Applied to the definition given above, the temporal Moran's  $I$  is formally defined as shown in Equation 2.2.

$$LISAT(v_a) = \left( \frac{v_a - m}{S} \right) \left[ \left( \frac{1}{(N + T + NT)} \right) \left( \sum_{n=1}^N \left( \frac{v_n - m}{S} \right) + \sum_{t=1}^T \left( \frac{v_{at} - m_t}{S_t} \right) + \sum_{n=1}^N \sum_{t=1}^T \left( \frac{v_{nt} - m_t}{S_t} \right) \right) \right]$$

Equation 2.2: Temporal LISA (source: [13])

Here,  $T$  is the number of past values included and  $t$  represents the  $t^{\text{th}}$  measurement before the current one. Accordingly,  $v_{at}$  is the value measured at node  $a$   $t$  measurements before the current one and  $m_t$  and  $S_t$  are the mean and the standard deviation of the whole population at time  $t$ . The resulting values are to be interpreted analogously to spatial LISA.

### 2.1.4 Statistical Significance - Monte Carlo Simulation

While the LISA statistics provide information on a measured value in relation to its neighbouring measurements, they do not allow statements on the statistical significance of this information. For this purpose, a statistical test has to be conducted. In this thesis, a Monte Carlo simulation is used to test a LISA value against a null hypothesis of complete spatial randomness (CSR).

In detail, a number of  $L$  distinct sets of random neighbours are chosen for each calculated LISA value  $LISA(v_a)$ . Thereafter, for each of these sets the LISA value is calculated, resulting in a set of random LISA values for  $v_a$ . This set is then ordered into a sequence which represents the sample probability distribution, and consequently the significance level for  $LISA(v_a)$  can be deduced by comparing  $LISA(v_a)$  to this probability distribution [14].

The formal definition of this simulation for spatial LISA is given in Equation 2.3 where  $l \in L$  represents a concrete set of random neighbours,  $n^{(l)}$  represents the random neighbours and  $N$  is the number of random neighbours for  $l$ .

$$LISA^{(l)}(v_a|CSR) = \left( \frac{v_a - m}{S} \right) \left[ \sum_{n=1}^N \left( \frac{1}{N} \right) \left( \frac{n^{(l)} - m}{S} \right) \right] \quad l = 1, \dots, L$$

Equation 2.3: Monte Carlo Simulation for Spatial LISA (adapted from [13])

For the temporal LISA variant, two different variations of Equation 2.3 are used. In the first variation, the set of  $N$  random neighbours is extended with  $T$  random past values from a node's actual neighbours. This variation allows for quick calculation without knowing the complete past network state, while still providing statistical significance for calculated values.

$$LISAT^{(l)}(v_a|CSR) = \left( \frac{v_a - m}{S} \right) \left[ \left( \frac{1}{(N + NT)} \right) \left( \sum_{n=1}^N \left( \frac{v_n^{(l)} - m}{S} \right) + \sum_{n=1}^N \sum_{t=1}^T \left( \frac{v_{n_t}^{(l)} - m_t}{S_t} \right) \right) \right]$$

Equation 2.4: Monte Carlo Simulation for Temporal LISA

The second variation considers the complete past network state, in that it extends the set of  $N$  random neighbours with random past values from the complete network. The formal definition shown in Equation 2.4 is analogous for both variations. However, for the first variation  $v_{n_t}^{(l)}$  is chosen from the measurements at time  $t$  of the actual neighbours of  $v$ , whereas for the second variation, it is chosen from all node's measurements at time  $t$ .

## 2.2 *Apache Spark and Spark Streaming*

In recent years, the *MapReduce* programming model, and subsequently its various implementations and surrounding platforms (e.g. *Apache Hadoop*), have received much attention both in science and in commerce. However, while *Apache Hadoop* provides a viable solution to many problems, recent work has shown various limitations with the programming model itself as well as with its field of application [16] [19] [17]. More specifically, applications which "reuse a working set of data across multiple parallel operations"[22] cannot be efficiently implemented by using *MapReduce*.

To address this issue, researchers at the University of Berkeley created *Spark*, an open-source, in-memory cluster computing platform. This platform specifically targets applications which run jobs iteratively or provide interactive query interfaces. Through in-memory data processing *Spark* substantially enhances the performance of *MapReduce*-like applications without sacrificing scalability or fault-tolerance [22].

The general-purpose data abstraction model introduced by *Spark* (cf. Section 2.2.2) furthermore allowed the platform to be expanded to suit different use cases. In particular, the need for a real-time data processing engine with a high-level interface was addressed by creating *Spark Streaming*. Building on the *Spark* platform, this component uses an interval-based programming model to provide real-time processing capabilities while assuring fault tolerance, consistency and integration with batch-processing [4].

The *Spark* project was adopted as an *Apache Incubator* project in 2013 and was promoted to a top-level project in 2014. Consequently, the name of the platform was changed to *Apache Spark*.

### 2.2.1 *Spark Architecture*

The *Spark*-platform is implemented in Scala and is hence run within the JVM. It can be deployed as a standalone application on a cluster, as a client application of either *Apache Hadoop 2+* (YARN) or *Apache Mesos*, or to a simulated cluster on a single machine (hereafter referred to as local master). In addition to a Scala API, interfaces in Java and Python are available. As the Scala API was used for the work presented in this thesis, the additional APIs are not discussed.



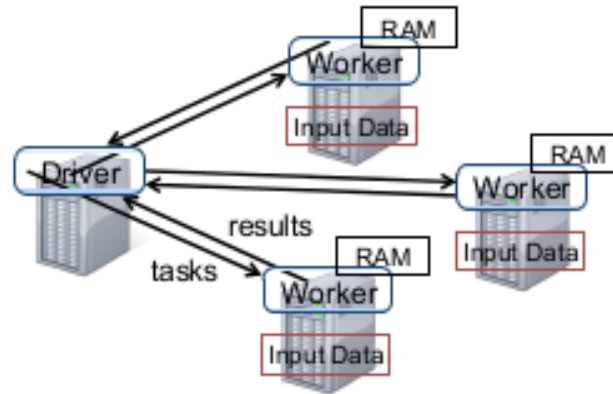


Figure 2.1: Spark Program Architecture (source: [21])

*Spark* provides two options for running applications. Firstly, an interpreter based on the included interpreter in the Scala language distribution allows users to interactively run queries on large data sets through the *Spark* engine. Secondly, applications can be written as Scala programs called *driver programs* and can be submitted to the cluster's master node after compilation.

Independently of the option chosen, the program runs on a cluster which is based on a configuration object called `SparkContext` defining the connection to the *Spark* master node. The master node, or driver, schedules tasks at the worker nodes, which submit their results back to the driver after task completion (depicted in Figure 2.1).

### 2.2.2 Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs) are the main data abstraction concept used in *Spark*. RDDs are formally defined as "read-only, partitioned collection[s] of records"[21]. More specifically, data from a persistent source is used to generate RDDs, which provide operations for data manipulation (so-called "transformations" [21]) such as `map()`, `filter()` or `join()`. Due to the read-only nature of RDDs (immutability), these transformations generate new RDDs (so-called child RDDs) instead of altering the data itself.

In a *Spark* application, usually many transformations are linked to calculate results. An RDD contain this sequence of transformations (forming a directed, acyclic graph (DAG), called "lineage" in *Spark*) which is used to calculate its data partitions from the

original data. The lineage graph is used to regenerate an RDD after a failure, ensuring fault-tolerance within an application. In addition to the lineage, RDDs contain meta data such as information on the location of their data partitions.

Moreover, RDDs also provide operations called "actions" which yield one or several values as a result. These actions include `count()`, `foreach()` and `saveAsTextFile()`, among others. Commonly, these actions are used to store data either on a file system, or in an arbitrary destination using `foreach()` [21].

### 2.2.3 Dependencies and Scheduling

As stated above, each RDD consists of a set of data partitions and lineage, along with a function to compute its elements based on the lineage and information on data location. Lineage information is actually stored as a set of dependencies to parent RDDs. The creators of *Apache Spark* distinguish two types of dependencies, namely *narrow* and *wide* dependencies. As shown in Figure 2.2, narrow dependencies occur if "each partition of the child RDD depends on a constant number of partitions of the parent" [21].

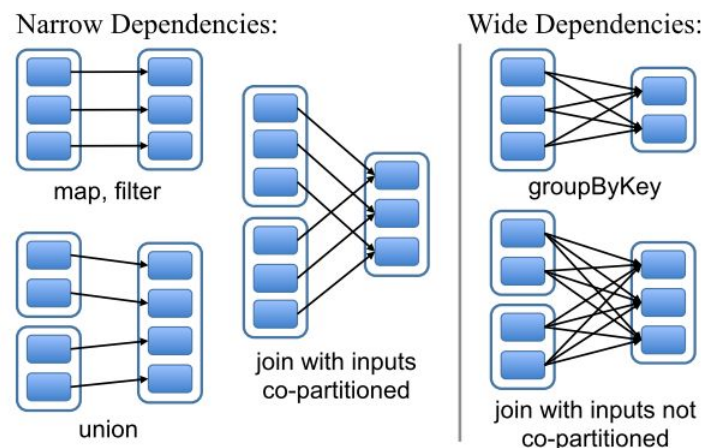


Figure 2.2: Spark Dependencies (source: [21])

Operations creating narrow dependencies include `map()`, `filter()` and `union()`. Wide dependencies, on the other hand, are created by transformations in which the result is dependent on a large part of the parent's RDD's partitions (possibly all of them). Examples for such transformations are `groupByKey()` and `join()`.

How an RDD depends on its parent is also important in terms of job scheduling, as it dictates how transformations can be distributed to cluster workers. RDDs created by a

transformation with narrow dependencies permit that each partition can be computed independently of other partitions on a cluster worker. In case of wide dependencies, on the other hand, "data from all parent partitions [is required] to be available and to be shuffled across the nodes using a MapReduce-like operation" [21]. The scheduler used by *Apache Spark* considers this information as depicted in Figure 2.3. It groups transformations into stages, each of which contains as many transformations with narrow dependencies as possible. Stages are delimited by transformation with wide dependencies. This allows to distribute easily parallelisable task (i.e. narrow dependencies) efficiently [21].

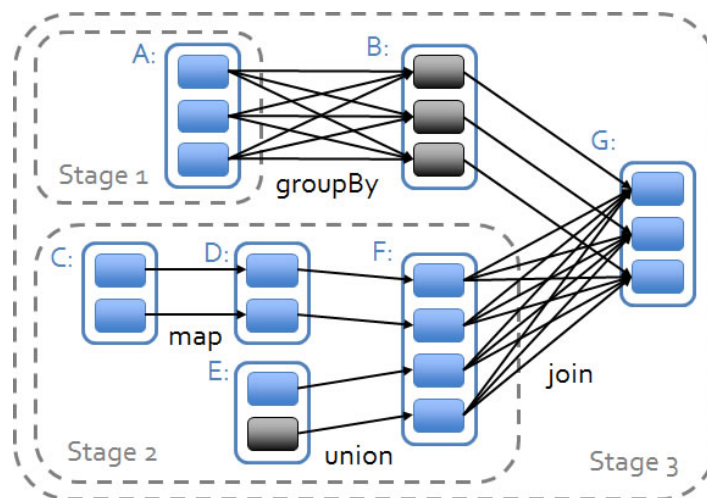


Figure 2.3: Spark Scheduling Stages (source: [21])

#### 2.2.4 *Spark Streaming*

As an extension to the *Apache Spark* batch processing platform, the creators of *Apache Spark* implemented a stream processing engine called *Spark Streaming*. For processing the input streams *Spark Streaming* uses an approach which differs from many existing streaming engines (such as Storm or Samza). While these systems are event based, and hence process each record in real time, *Spark Streaming* uses a "micro-batch" [15] [7]. More specifically, *Spark Streaming* captures input from a stream for a pre-defined interval. At the end of the interval, it creates a batch upon which data manipulation operations are performed. Each batch is stored as a set of RDDs, which allows the data to be processed by using the *Apache Spark* engine. An API, called Discretised Stream (DStream), for manipulating data on such streams of RDDs is provided by *Spark Streaming*.

### 2.2.4.1 Discretised Streams

Analogously to RDDs in *Apache Spark*, *Spark Streaming* programs define transformations and actions on DStreams to manipulate and store data read from an input stream. Several connectors to input sources are provided by the DStream API, along with several types of transformations and output actions.

A distinction can be drawn between two types of transformations available on DStreams. There are transformation such as `map()` or `filter()` which can be applies to DStreams of any type, whereas some transformations can only be applied to DStreams which contain key-value tuples. `join()`, `groupByKey()` and `reduceByKey()` are examples of such transformations.

All transformations described above are stateless, meaning that for each batch interval only the data collected in this interval is considered. However, *Spark Streaming* also provides several ways of making transformations stateful. Firstly, "windowing" a DStream creates a sliding window over several batch intervals. For example, calling `window(Seconds(5))` on a DStream in a job with a batch duration of one second will yield a DStream containing, at each batch interval, input data of the last five seconds. Secondly, data can also be aggregated over time by using either more specialised forms of windowing such as `reduceByWindow()` or the `updateStateByKey()` transformation [23].

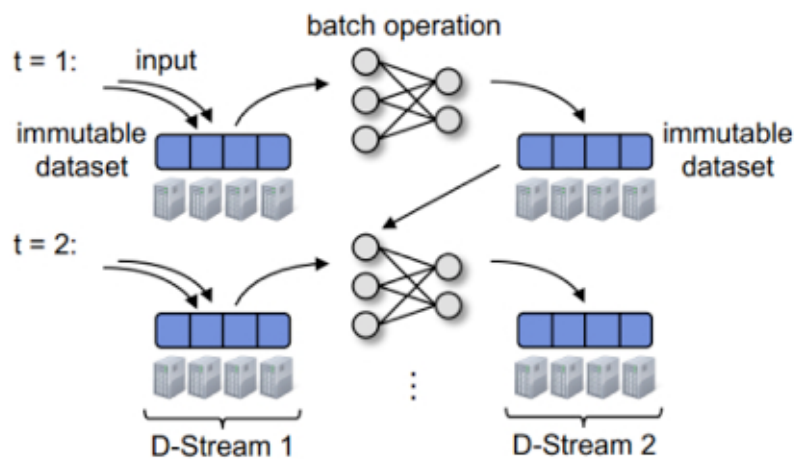


Figure 2.4: DStream Processing Model (source: [23])

Figure 2.4 depicts the programming model described above. In particular, two subsequent batches of the same program are shown. At  $t = 1$ , the input dataset is transformed using stateless transformations. The resulting dataset is included in the transformations at  $t = 2$  by using a stateful transformation.

#### 2.2.4.2 Data Input

As stated above, there are several connectors for input streams already included in *Spark Streaming*, namely connectors for *Apache Kafka*, *Apache Flume*, *Twitter*, *ZeroMQ*, *Amazon Kinesis* and *MQTT*. Furthermore, connectors for reading from file systems and for listening on a socket are included. However, the API of *Spark Streaming* also allows for creating custom input stream sources, called receivers.

The receiver API provides methods to repeatedly push data to a *Spark Streaming* application. *Spark Streaming* collects the data and processes it according to the driver application at each batch interval. One or multiple instances of such receivers can be registered with an application, each of which provides a separate DStream within the driver program. At application start-up, *Spark Streaming* creates the instances and registers them as tasks at different cluster nodes [3].

### 2.2.5 Spark Streaming vs. Apache Storm

Stream processing systems which target Big Data use cases have become more and more popular in recent years. Most notably, *Apache Storm* has gained much attention as it greatly simplifies both configuration and application programming compared to earlier real-time stream computation approaches [9]. In an earlier master thesis, Simpal Kumar used *Apache Storm* to compute LISA statistics in the context of WDN [18]. In order to differentiate this thesis from the work presented in [18], the following section gives a brief overview of the similarities and differences between *Apache Storm* and *Spark Streaming*.

In terms of application management, the two systems are similar. In particular, both systems use a single application master node, called *Nimbus* in *Apache Storm* and driver node in *Spark Streaming*. In addition, in both systems cluster workers receive parts of the application code to run. However, in *Apache Storm* state management and coordination between the *Nimbus* and the worker program instances (called *Supervisors*) is handled by *Zookeeper*. Hence, both the *Nimbus* and the *Supervisors* are stateless while the driver

node handles state in *Spark Streaming* [10].

In contrast to the cluster management, the stream processing is handled differently in the two systems. *Apache Storm* processes tuples of input values in an event based fashion, so that each arriving input value is processed as it enters the program, whereas *Spark Streaming* uses a micro-batch approach as described in Section 2.2.4.

The programming model offered by *Apache Storm* is comparable to some extent to the one offered by *Spark Streaming*. Programs written for *Apache Storm* define *Topologies* which consist of input sources (*Spouts*) and operations (*Bolts*), and express a graph of operations which is traversed by each tuple of input values. *Spouts* are similar to receivers in *Spark Streaming* and *Bolts* correspond to DStream operations. However, the way in which operation parallelism is achieved is different in the two systems. In *Spark Streaming*, the master node distributes Scala closures to workers, which process data simultaneously (data parallelism [6]). *Apache Storm*, on the other hand, creates tasks from *Bolts* and distributes these on the cluster to run them simultaneously (task parallelism [11]) [8] [5].

Consequently, aggregation operations cannot be done as simple calls in *Apache Storm*. Instead, *Bolts* have to be registered with so-called field groupings. Thereby, tuples with equal values in one field are distributed to the same worker task which in turn enables aggregation operations.

In conclusion, while *Spark Streaming* and *Apache Storm* offer similar capabilities for stream processing, their architectures differ considerably. Most notably, *Apache Storm* uses a record-at-a-time processing model, whereas *Spark Streaming* processes stream data as micro-batches, which leads to different programming models.

## 2.3 Anomaly Detection in Water Distribution Networks

As already mentioned, the fundamental framework of this thesis is a WDN scenario described in [13]. This scenario covers a complete architecture for anomaly detection in the context of WDN, including a multi-layered hardware architecture as well as the according software components. It also shows how LISA statistics can be applied in the context of a continuous-time setting.

The base layer of the hardware architecture consists of sensors deployed at the network nodes, i.e. "pipe junctions and network end points where water is extracted for consumption" [13]. These sensor are simple electronic devices designed for durability and energy efficiency and are consequently limited in terms of functionality. More specifically, they

only need to measure particular characteristics of the WDN and to periodically transmit these measurements to intermediate computing stations.

These so-called base stations serve several purposes. Apart from collecting sensor measurements, they calculate local LISA statistics. Furthermore, they are interconnected and form an overlay network in which measurements and detected anomalies are shared. Finally, an ADBMS stores all sensor measurements and anomalies, and provides means for more complex analytic operations. In particular, LISA statistics can be calculated globally by using data from the complete network [13].

# 3

## Implementation

To evaluate if and how the original scenario described in [13] can be applied to *Spark Streaming*, several prototypes for different algorithms have been developed in the course of this thesis. This chapter covers the general architecture of these prototypes as well as some abstractions and simplifications made from the original implementation. Furthermore, the core algorithms of the implementation, which calculate the statistical measures described in Section 2.1, are explained in detail.

### 3.1 Architecture

Figure 3.1 shows an overview of the application architecture used in all scenarios which are explained in the following. Based on information on a simulated WDN topology sensor values (key-value pairs in the figure) are generated by a *Spark Streaming* receiver.



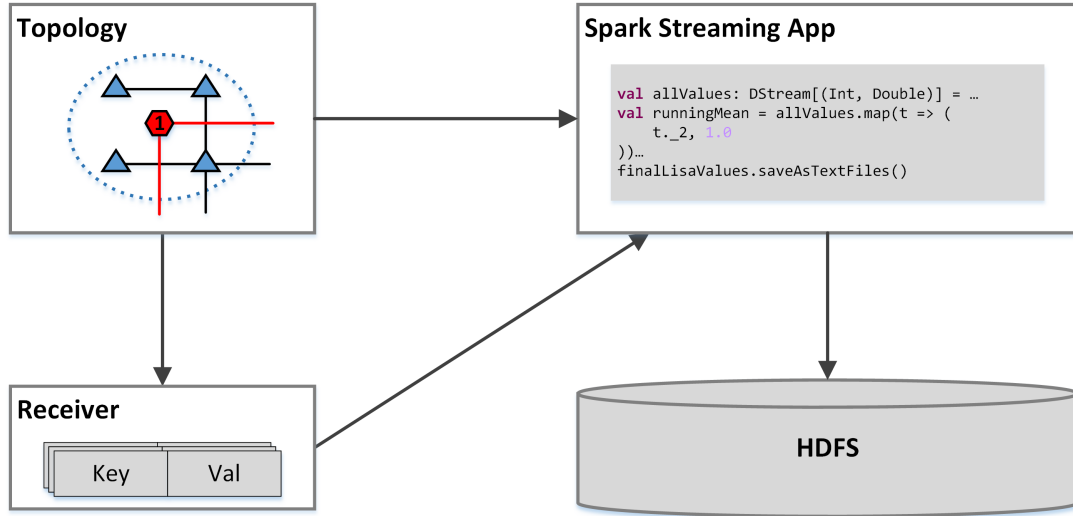


Figure 3.1: Application Architecture Overview

The *Spark Streaming* driver application thereafter processes these sensor values according to one of the algorithms covered in Section 3.2. Finally, input values and calculation results are stored to a *Hadoop File System (HDFS)*. Each of these processes is detailed in one of the following sections.

### 3.1.1 Network Topology

LISA indicators are calculated based on relationships between measurements at different sensors in a WDN, as described in Section 2.1. In [13], they were applied to sensor data captured from a real-world network as well as to data from a simulated network. The structure of this simulated network is used in all network topologies in this thesis.

More precisely, all topologies used in this thesis are generated, quadratic, grid-like networks. Hence, nodes can only be connected to a maximum of four other nodes. The side-length of the grid is equal to the square of the number of nodes in the topology. A simplified example consisting of 16 nodes is shown in Figure 3.2.

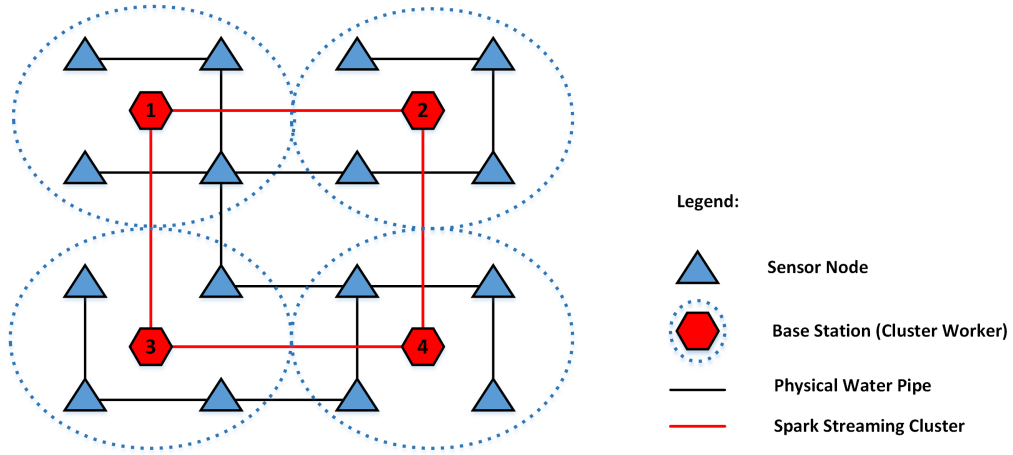


Figure 3.2: Simple Topology Model (adapted from [18])

Several different topologies were generated in order to test the algorithms (cf. Section 4) differing both in terms of the number of sensor nodes and the density (i.e. number of connections between nodes). For this purpose, simple proximity matrix files which list all connections for each node were generated by using a *Python* script. The content of such a file, representing the topology in Figure 3.2, is shown in listing 1. In order to use this information in a *Spark Streaming* program, the files were converted into an XML representation at program initialisation.

Moreover, the XML also contains a second network component, namely so-called base stations. In the real-world network, these are computing stations which collect sensor measurements from a group of sensors through wireless connections. They are interconnected and form a stream-processing subsystem. In this implementation, however, nodes in a cluster running *Spark Streaming* are used as stream-processing system, which is also shown in Figure 3.2. As a distinction to sensor nodes, they are subsequently referred to as cluster workers.

```

1  0100000000000000
2  1000010000000000
3  0001000000000000
4  0010000100000000
5  0000010000000000
6  0100101001000000
7  0000010100000000
8  0001001000000000
9  0000000000001000
10 0000010000100000
11 0000000001010010
12 0000000000100001
13 00000000010000100
14 0000000000001010
15 0000000000100100
16 0000000000010000

```

Listing 1: Sample Proximity Matrix File

### 3.1.2 Receiver

As stated above, all network topologies used in this thesis are artificial. As no actual sensor data was available for the *Spark Streaming* program the data needed to be simulated. In order to mimic the behaviour of sensors and base stations in a real-world WDN, simulated values for separate parts of the network are collected at each cluster worker (as shown with dotted blue areas in Figure 3.2).

As described in Section 2.2.4.2, *Spark Streaming* provides an API for writing custom input sources, called receivers. While this receiver API is quite well suited to push simulated sensor data into a *Spark Streaming* application, it is not possible to control at which cluster worker a receiver instance will be registered. As they are registered as *Spark* tasks, it is even possible that multiple receivers may be assigned to a single worker. This behaviour may have negative impact on the performance of an application, as task slots are occupied by receivers for the whole run time of an application.

Consequently, recreating the exact application structure described in [13] using *Spark Streaming* was not feasible. Instead, a single receiver instance for simulated sensor values was used for all algorithms. There are two different variations of the receiver implementation for spatial and temporal LISA calculations. In case of spatial LISA, the receiver emits tuples containing an integer as sensor node key and a simulated sensor value. In case of temporal LISA, instead of a single value, an array of length  $k + 1$  containing one new value as well as the  $k$  last values is emitted. Here,  $k$  is a parameter configurable for each application run.

In the real-world topology, a sensor measures a number of different metrics, such as water quality and pressure. As the presented implementation focuses on the calculations and the performance of *Spark Streaming* rather than on the one-to-one applicability to the real-world scenario, only values for a single artificial metric are emitted by each sensor. More specifically, for each sensor node random values distributed according to a Gaussian function with Mean 0.0 and Standard Deviation 1.0 are generated at the receiver.

The rate at which these values are pushed into the application has been adapted to the characteristics of *Spark Streaming*. To ensure correctness of the LISA calculations, exactly one value per sensor and batch duration is generated. Consequently, the sensor rate is aligned with the batch duration of an application run. This issue is discussed in further detail in Section 4.

### 3.1.3 Output Processing

The DStream API provided by *Spark Streaming* offers a simple output function which is used for all output, namely `saveAsTextFile()`. For the algorithm described subsequently, this function is used to store both input values and final results to a HDFS location. Several Python scripts are used to validate the calculated results, and to parse the log files written by *Spark Streaming* for performance evaluation.

## 3.2 Algorithms

The following section details the implementation of the various LISA and Monte Carlo algorithms with *Spark Streaming*. Important parts of all algorithms are depicted as graphs, which consist of the elements shown in the legend in Figure 3.3. All graphs show sequences of operations (depicted as connectors) on DStreams (blocks).

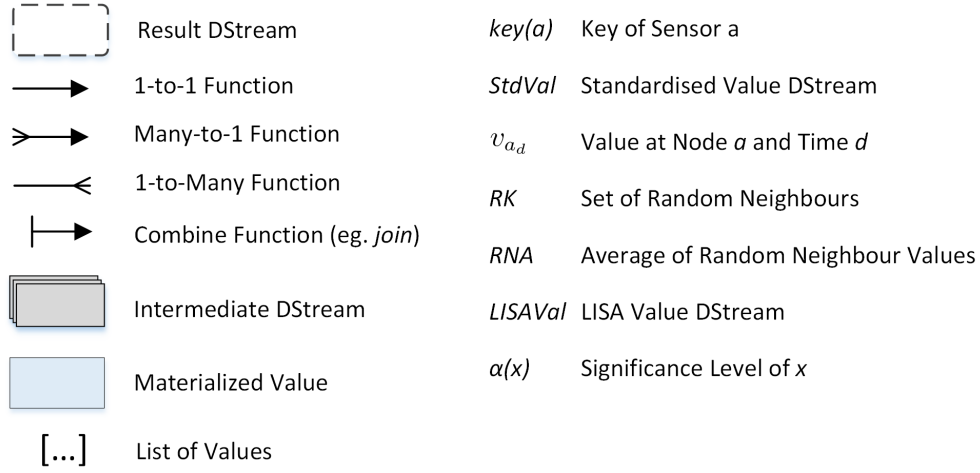


Figure 3.3: Legend

### 3.2.1 Spatial LISA

The input of the LISA algorithm consists of key-value pairs in the form  $(key(a), val_a)$  for each sensor  $a$  emitted by the custom receiver described above. As explained in Section 2.1, all values need to be standardised by using the form  $\frac{(Value - Mean)}{StandardDeviation}$ . Consequently, as shown in Figure 3.4, calculating both the mean and the standard deviation are the first steps in the algorithm, where the resulting DStream (*Mean DStream* ( $M$ )) from the calculation of the mean value is used to calculate the standard deviation (*Standard Deviation DStream* ( $S$ )). In a next step the standardised values (*StdVal*) are calculated by mapping  $M$  as well as  $S$  to each value of the initial DStream by using a simple *map()* transformation.

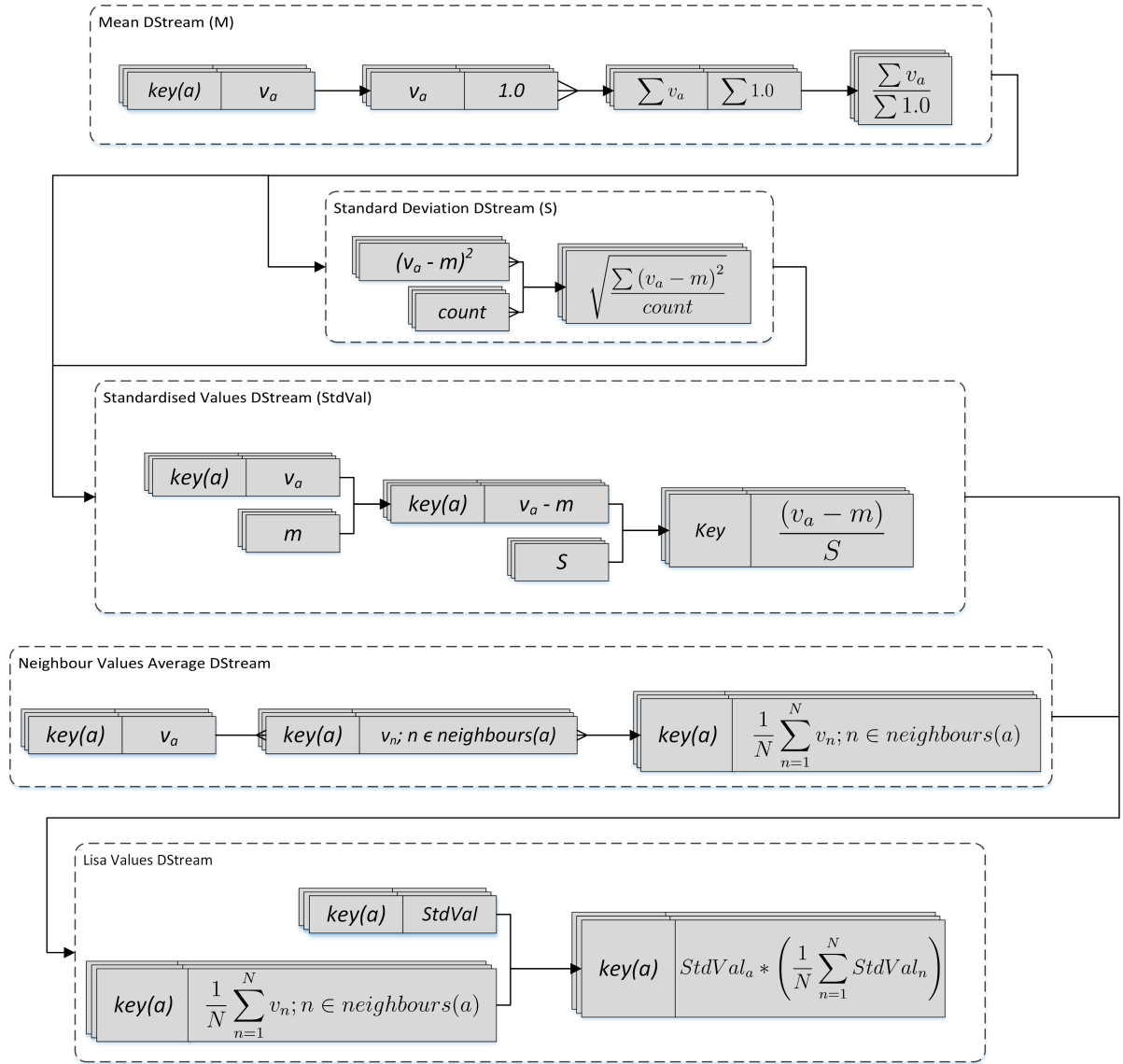


Figure 3.4: Spatial LISA Algorithm

The calculation of the average of all standardised neighbour values requires two steps. Firstly, each value is mapped to all neighbouring sensors' keys. As a result, the values of all according neighbours are mapped to each key. In a second step, a *groupByKey()* transformation can be used to collect these values and calculate their average for each key (*Neighbour Values Average DStream*). The resulting DStream is joined to the *StdVal* DStream in order to calculate the LISA values (*LISA Values DStream*).

### 3.2.2 LISA with Temporal Association

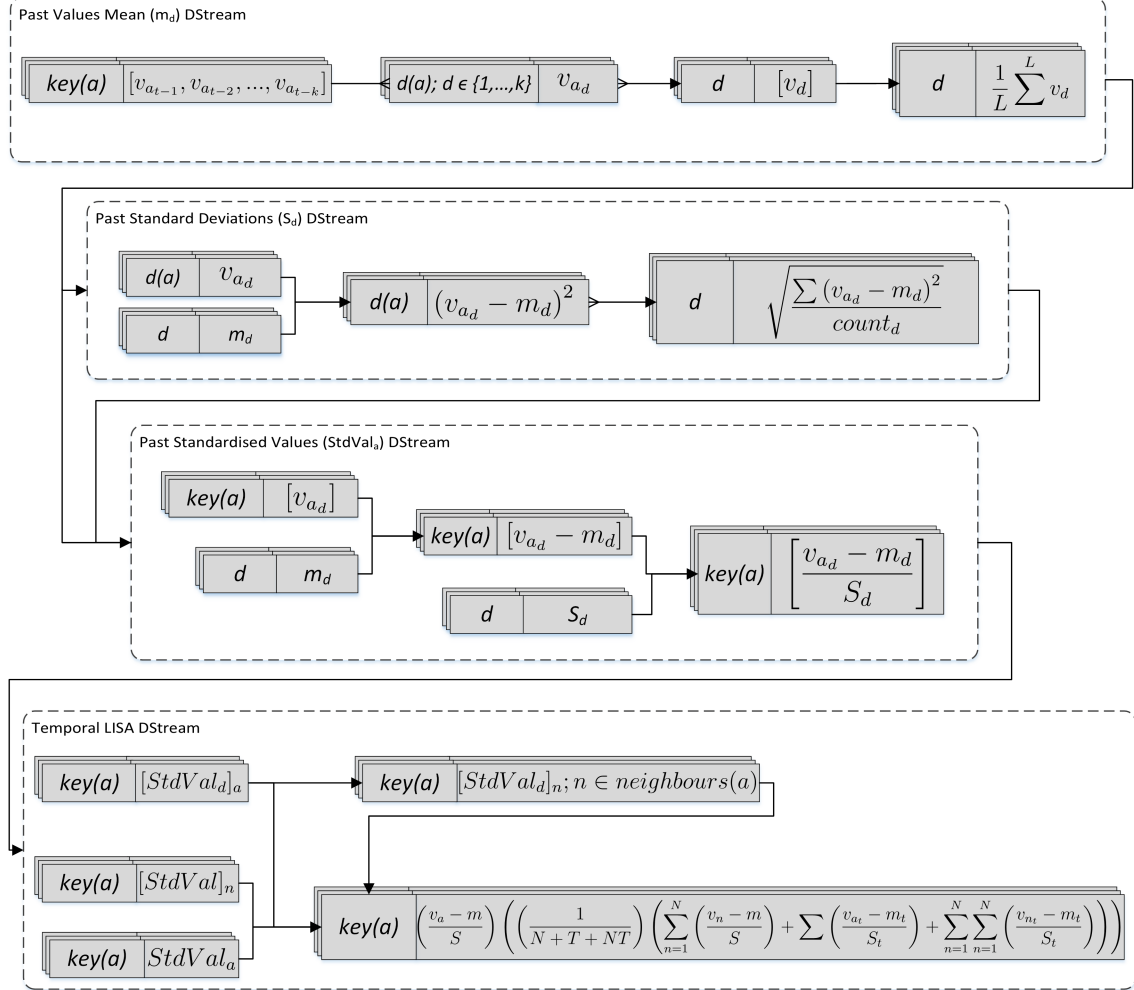


Figure 3.5: Temporal LISA Algorithm

For the temporal variation of the LISA algorithm, described in Section 2.1.3, the receiver used for the spatial LISA algorithm was modified to emit a list of values for each key, containing a new ("current") value along with  $k$  past values.  $S$ ,  $m$  and  $StdVal$  for current values are calculated analogously to the spatial LISA algorithm. To calculate the mean and the standard deviation for each past network state  $d \in 1, \dots, k$ , all values are mapped to the according state  $d$ . Thereafter, calculating the standardised values is done analogously to the spatial LISA algorithm for each  $d$ .

As the current as well as the past neighbour values are necessary for calculating the temporal LISA values (cf. Figure 2.2), the mapping of the neighbours' keys described in the previous sections is also applied to every  $d$ . Finally, all resulting DStreams are combined to calculate the results.

### 3.2.3 Spatial Monte Carlo Simulation

As a statistical significance test for spatial LISA values, the spatial Monte Carlo Simulation algorithm is built to complement the spatial LISA algorithm. The standardised values DStream ( $StdVal$ ) is used as input. For this thesis, due to the poor performance of the algorithm created initially, an improved variation was created. While the improved version should be favoured over the initial version in any further work, the initial version is described for reference and in order to highlight certain properties of *Spark Streaming* (cf. Section 4.5).

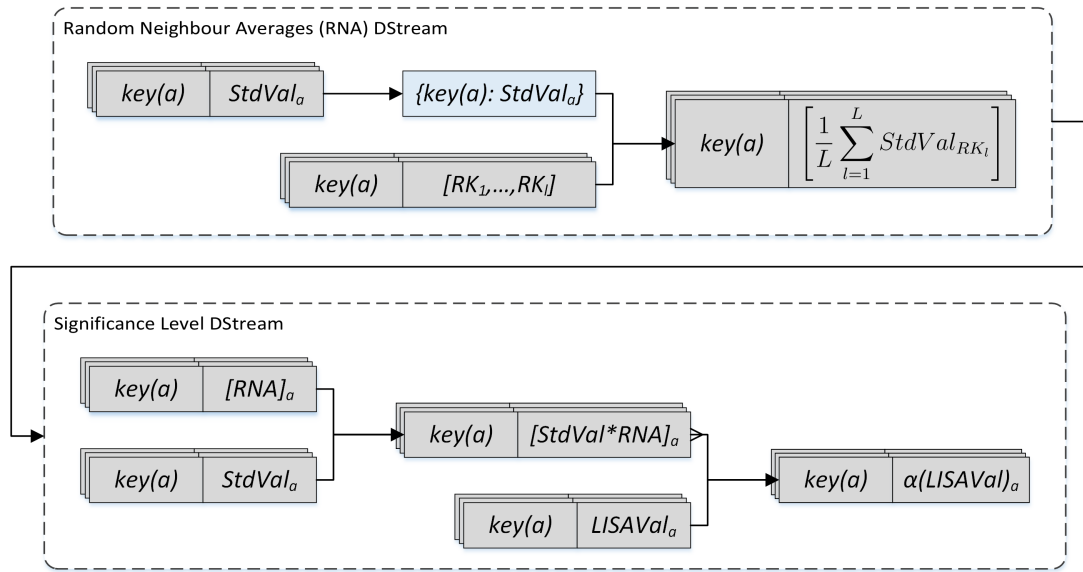


Figure 3.6: Spatial Monte Carlo Simulation - Naive Approach

Along the standardised value DStream, a second DStream is used as input. This DStream is created by a custom receiver, which emits 1000 random neighbour sets for every batch and for each node. More specifically, these sets consist of one up to four distinct random node keys representing neighbours. Furthermore, the sets are unique for



every batch and node i.e. they are distinct combinations of node keys.

The two variations of the algorithm mentioned above differ in the way one of the intermediate results is calculated, namely the averages of the random neighbour values. In both versions, firstly all standardised values of a batch are materialised at one node (blue box in figures 3.6 and 3.7). In the initial version of the algorithm, the resulting map is thereafter mapped to each of the random neighbour key sets, allowing the averages of the random neighbour values to be calculated directly. In the improved approach, however, the resulting map is mapped to each of the node keys. The resulting DStream thus contains the complete key-value map in this batch at each node key, which allows all subsequent calculation to run in parallel without dependencies between DStream partitions. To calculate the average neighbour values, this DStream is thereafter joined with the DStream containing the random neighbour key sets.

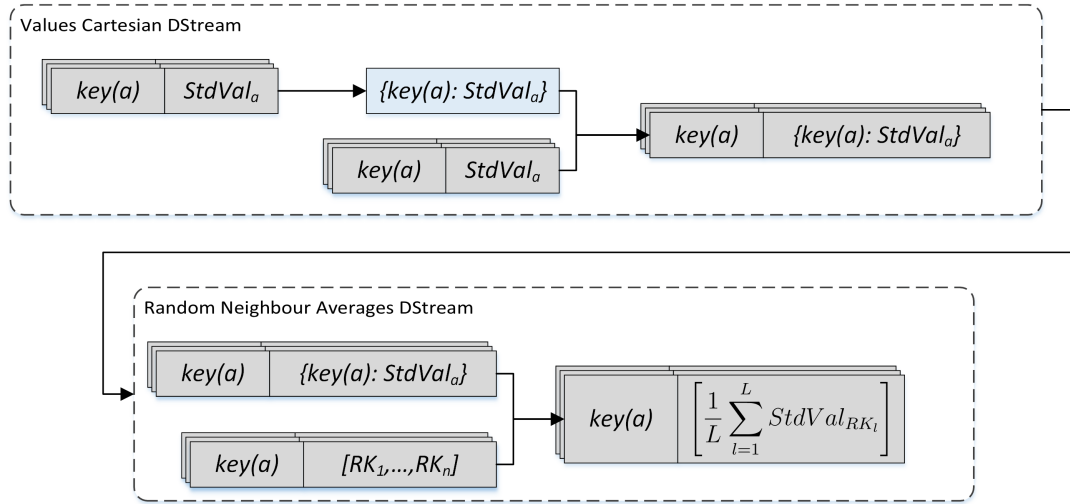


Figure 3.7: Spatial Monte Carlo Simulation - Improved Approach

Finally, in both versions of the algorithm the *Random Neighbour Averages (RNA)* DStream is joined with the initial input DStream and with the *LISAVal* DStream resulting from the spatial LISA algorithm in order to calculate the random LISA values. To obtain the significance level for these LISA values, the hypothetical position of the values in the according sequence of random LISA values is used. For example, if a LISA value would be entered in the sequence at position 996, the significance level  $\alpha$  for this LISA value would be  $\alpha = \frac{996}{1001} \approx 0.995$ , or 99.5%.

### 3.2.4 Temporal Monte Carlo Simulation

The most advanced algorithm implemented in this thesis applies a Monte Carlo Simulation to the temporal variant of LISA statistics. Therefore, standardised values from random neighbours are sampled among both past and current network states. As proposed in [13], two approaches of the algorithm were implemented. In the first version, random values are sampled from the complete network state for each node ("global selection"). In order to reduce traffic between cluster workers, as well as to reduce the overall calculation complexity, the second version of the algorithm only samples random values from node neighbours ("local selection").

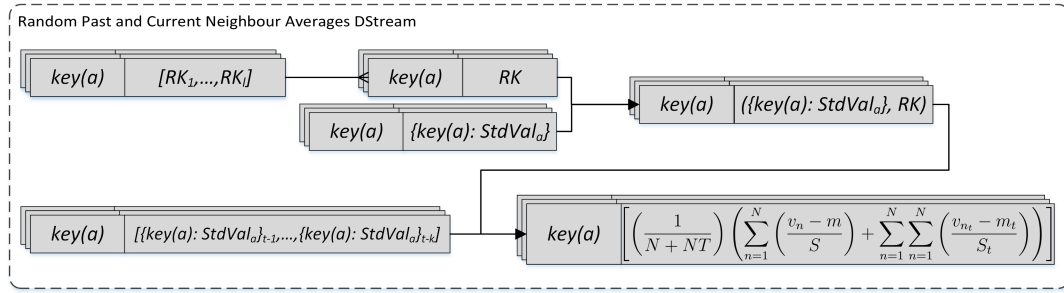


Figure 3.8: Temporal Monte Carlo Simulation

Both the global and the local selection algorithm are based on the improved version of the spatial Monte Carlo Simulation described above and work analogously except for the selection of random past neighbour values. As shown in Figure 3.8, at first the DStream of lists of random neighbour sets (cf. Section 3.2.3) is split into single sets each of which is attributed to a key. Thereafter, the result is joined to the DStream containing all current node key-value pairs per node key, which is constructed as shown in Figure 3.7. The resulting DStream is in turn joined with the selected past neighbours values, allowing to calculate the list of average random neighbour values for each node key. Finally, the random LISA values, and subsequently the significance levels for the measured values, are obtained analogously to the spatial Monte Carlo Simulation described in the previous section.

```

1 val randomNeighbourAvgs = allRandomLisaValues.map(t => {
2   val pastValuesSize = t._2._2.values.head.size
3   val filteredMap = t._2._2.filter(me => me._1 != t._1)
4
5
6   val randomPastValues: List[Double] = (for (i <- 0 until pastValuesSize) yield
7     (for (_ <- 1 to new Random().nextInt(4)+1) yield
8       new Random().shuffle(filteredMap.values).head(i)).toList).toList.flatten
9
10  val randomCurrentValues: List[Double] = t._2._1._1
11    .filter(me => t._2._1._2.contains(me._1)).values.toList
12  val allRandomValues: List[Double] = randomCurrentValues ++ randomPastValues
13  (t._1, allRandomValues.foldLeft(0.0) (_+_)) /
14    allRandomValues.foldLeft(0.0) ((r,c) => r+1))
15 })

```

Listing 2: Global Selection of Random Past Neighbour Values

Listing 2 shows the code for the calculation of the random neighbour averages (final DStream in Figure 3.8). A `map()` transformation is applied to each key-value pair in the DStream containing all past standardised values as well as one up to four randomly selected current standardised values (line 1). In lines 6 to 8, past standardised values are selected randomly from the global network state. More specifically, for each past network state `i`, a set of random values is selected from the variable `filteredMap`. The number of past standardised values is randomly chosen between one and four. `filteredMap` consists of all past standardised values, excluding values for the node for which the significance level is to be calculated.

```

1 val randomNeighbourAvgs: DStream[(Int, Double)] = allRandomLisaValues.map(t => {
2   val pastValuesSize = t._2._2.values.head.size
3
4   val randomPastNeighbours: List[List[Int]] =
5     (for (rk <- 0 until pastValuesSize) yield
6       (for (_ <- 1 to new Random().nextInt(4)+1) yield
7         new Random().shuffle(nodeMap(t._1).getNeighbour.toList)
8           .head.substring(4).toInt).toList).toList
9
10  val randomPastValues: List[Double] =
11    (for ((n, idx) <- randomPastNeighbours.zipWithIndex) yield
12      (for (i <- n) yield t._2._2(i)(idx)).toList).toList.flatten
13
14  val randomCurrentValues: List[Double] = t._2._1._1
15    .filter(me => t._2._1._2.contains(me._1)).values.toList
16  val allRandomValues: List[Double] = randomCurrentValues ++ randomPastValues
17  (t._1, allRandomValues.foldLeft(0.0) (_+_)) /
18    allRandomValues.foldLeft(0.0) ((r,c) => r+1))
19 })

```

Listing 3: Local Selection of Random Past Neighbour Values

In contrast to the global selection algorithm, the local selection algorithm considers actual neighbouring nodes for the selection of past random neighbour values only. As shown in listing 3, the `map()` operation used is very similar to the global selection algorithm. However, instead of selecting values from the complete network, random sets of node keys are selected among the node's actual neighbours for each `i` (lines 4 to 7). Thereafter, values from this list are obtained according to these keys (lines 9 and 10, cf. Section 2.1.4).

# 4

## Experiments

This chapter explains the setting, the parameters and the results of the experiments conducted for each of the algorithms described in the preceding chapter. In particular, the test bed and input data is explained, along with performance measurements and the according interpretations for each of the scenarios examined.

### 4.1 Setting

#### 4.1.1 Test Bed

As a test bed for evaluating the performance of *Spark Streaming* on LISA statics calculation, a *Hadoop YARN* cluster of 16 nodes managed by *Cloudera Manager v5.1* was used. The specifications of the cluster nodes are shown in Table 4.1. For submitting applications to *Spark Streaming*, the `spark-submit` command with the parameter `--master yarn-cluster` was used, which initiates the *Spark* driver to run as an application master managed by *YARN* [1].

| Component | Node Type 1      | Node Type 2      |
|-----------|------------------|------------------|
| CPU       | i7-2600 - 2 core | i7-4770 - 4 core |
| RAM       | 32 GB            | 32 GB            |
| Harddisks | 1 x 500 GB       | 2 x 500 GB       |
| Quantity  | 12               | 4                |

Table 4.1: Cluster Node Specifications (adapted from [20])

### 4.1.2 Topology

For comparability, all experiments were conducted with the same topology (hereafter called "standard topology") where not mentioned otherwise. It consists of 1600 nodes, and forms a connected graph. As explained in Section 2.1, all connections in the graphs are assigned a weight of 1, and are hence treated as equal in the calculations.

### 4.1.3 Input / Output

The input data used for all experiments was continuously generated by a custom *Spark* receiver at a certain rate. Where not mentioned otherwise, the rate corresponds to the selected batch duration insofar as for every node, a single value per batch is generated. The values are generated randomly using a Gaussian distribution with a mean of 0.0 and a standard deviation of 1.0 (cf. section 3.1.2) [2].

During each experiment execution, all generated input values, as well as the according calculated output values, were stored on a HDFS location. Furthermore, for the evaluation of calculation durations, the log file of the *Spark* driver node was taken into consideration.

## 4.2 Spatial LISA Calculation

The first experiment conducted was the calculation of the local Moran's I, or spatial LISA, as described in Section 3.2.1. At a first stage, the LISA algorithm was run with the standard topology, at a rate of 20 values per second and an according window length of 3 seconds. The goal of this experiment was to show how the number of cluster nodes used influence the time needed for each batch to be calculated. Separate runs for one, two, four, eight and 16 cluster nodes were evaluated.

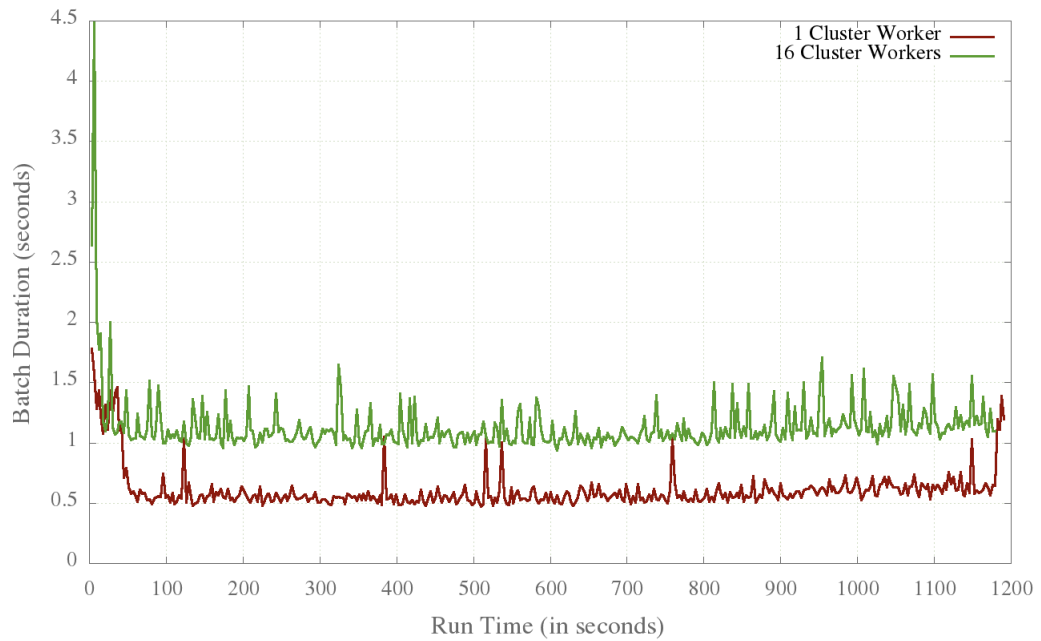


Figure 4.1: Single LISA Run for 1 and 16 Nodes

Figure 4.1 shows the batch durations in single runs for one (red) and 16 nodes (green). As is clearly visible, after an initial drop the calculation time remains more or less stable. Furthermore, the graphs shows that the calculation is slower with more nodes. More detailed evidence of this outcome is visible when comparing the average calculation time of all runs conducted, shown in Figure 4.2.

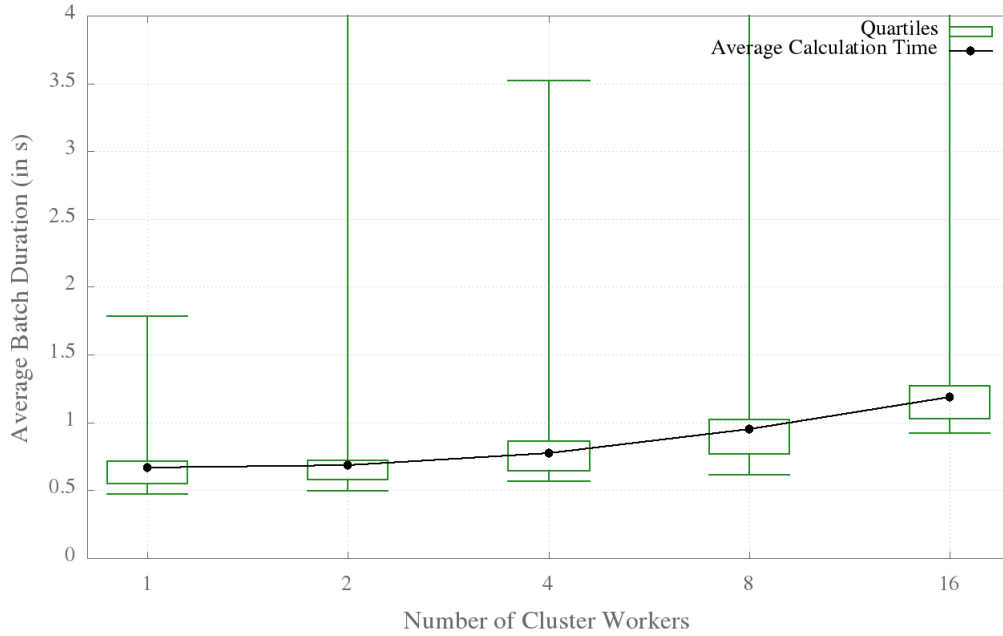


Figure 4.2: Average Calculation Times

One possible explanation for this behaviour lies in the output. As each worker has to save its calculated data, the overhead for writing to HDFS might cause a larger delay when more workers are involved. An analysis of the log files shows that indeed the time used to save results to HDFS increases proportionally to the number of workers, as shown in Table 4.2. While these differences in delays may explain part of the global difference, they cannot account for the total overhead caused by additional workers. This becomes apparent when comparing the two values side-by-side.

| Workers | Average Save Duration (in s) | Average Batch Duration (in s) |
|---------|------------------------------|-------------------------------|
| 1       | 0.1748 s                     | 0.6654 s                      |
| 2       | 0.1849 s                     | 0.6882 s                      |
| 4       | 0.2022 s                     | 0.7741 s                      |
| 8       | 0.2329 s                     | 0.9489 s                      |
| 16      | 0.2473 s                     | 1.1874 s                      |

Table 4.2: HDFS Save Duration per Number of Workers

To analyse the impact of the data volume, the experiment was repeated with higher load in a second stage. As the throughput could not be increased by scaling up the



input rate (cf. Section 2.2), different topologies were chosen to achieve this goal. More specifically, topologies with 3600, 10'000 and 25'600 nodes were chosen. The measured average calculation times are shown in Table 4.3.

| Workers | 1600 nodes | 3600 nodes | 10'000 nodes | 25'600 nodes |
|---------|------------|------------|--------------|--------------|
| 1       | 0.6654 s   | 0.8046 s   | 1.3034 s     | 2.2423 s     |
| 2       | 0.6882 s   | 0.8600 s   | 1.2756 s     | 1.9329 s     |
| 4       | 0.7741 s   | 0.9736 s   | 1.2400 s     | 2.0994 s     |
| 8       | 0.9489 s   | 1.0200 s   | 1.3746 s     | 2.4511 s     |
| 16      | 1.1874 s   | 1.2597 s   | 1.7245 s     | 2.7867 s     |

Table 4.3: Average Calculation Time per Number of Nodes and Cluster Workers

Furthermore, the evaluation of CPU and network usage support the findings described above. Figure 4.3 shows the CPU usage as stacked graph, i.e. the entire coloured area represent the cumulated CPU usage while the colours themselves represent single cluster worker CPU usage. The y-axis scale represents the cumulated percentage, where the complete CPU utilisation would be 1600% (as there are 16 cluster workers). The x-axis, represents the time as continuous stream. Two consecutive runs of the spatial LISA algorithm are shown in the figure; on the left a run with 25600 nodes and one cluster worker between the markers at 7:05 and at 7:11, and a second one with the same number of nodes and 16 cluster workers on the right between the markers at 07:12 and at 07:19 . As is clearly visible in the figure, CPUs utilisation is rather low (below an average of 5% per worker), and also unevenly distributed in the run with 16 workers.



Figure 4.3: Sample CPU Usage in the Cluster

The network utilisation for the same two runs is shown in Figure 4.4 in a similar fashion. Here, the dark blue area represents traffic received within the cluster network, while the light blue area represents traffic sent. The graph shows that in an initial phase, the traffic between workers reaches a peak. Thereafter, it recedes to a more or less steady level for the remainder of the run time. With 16 workers, the traffic remains much higher than with only one worker.

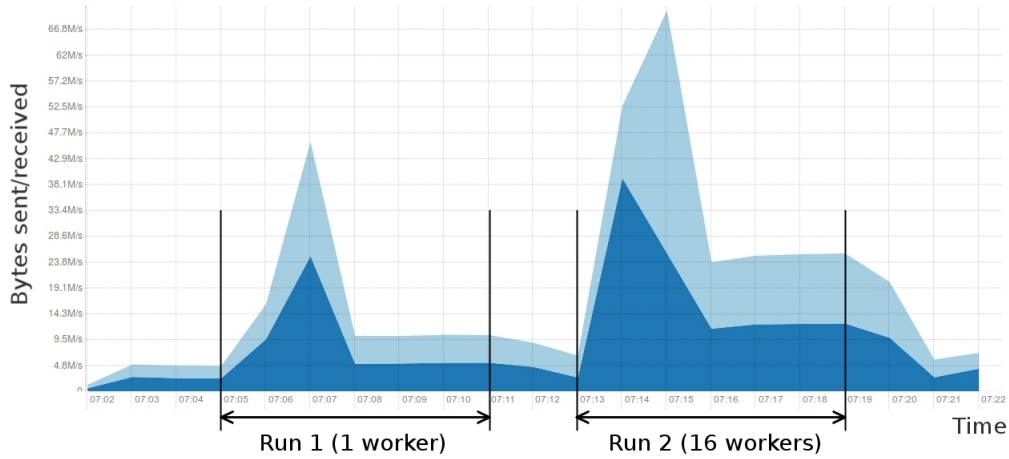


Figure 4.4: Sample Network Usage in the Cluster

In order to interpret these results, the structure of the local Moran's  $I$  and of the according algorithm (sections 2.1 and 3.2.1) have to be considered in detail. As depicted in Figure 3.4, both the mean and the standard deviation depend on the complete set of measurements taken during a batch interval. While the `reduce()` transformations used in the algorithm are partially parallelisable, all intermediate results have to be collected at the driver node, and redistributed to the workers for subsequent steps of the algorithm in order to produce the final results (e.g. the mean of all values). The increased network traffic in the run with 16 workers reflects this result.

In addition, the low CPU utilisation described above leads to a possible explanation of the performance behaviour. As the calculations in the algorithm are very simple, the overhead of parallelisation may outweigh the achieved gain in calculation performance from the additional cluster workers. Consequently, the behaviour is consistent independently of the load with which the algorithm is run.

### 4.3 LISA with Temporal Association

Similar to the first stage of the experiment described above, the algorithm for calculating LISA with temporal association was run on *Spark* for the standard topology, with a rate of 20 values per minute and a window duration of 3 seconds. In addition, the numbers of past sensor values to be included ( $k$ ) were chosen as 1, 2, 5 and 10. Figure 4.5 depicts the average calculation times measured for these  $k$  different values, including  $k=0$  (LISA without temporal association) as reference. Similar to the results described in the previous section, the calculation time does increase with a larger number of workers. In addition, the number of additional values in the calculation ( $k$ ) does not influence the performance in any significant way.

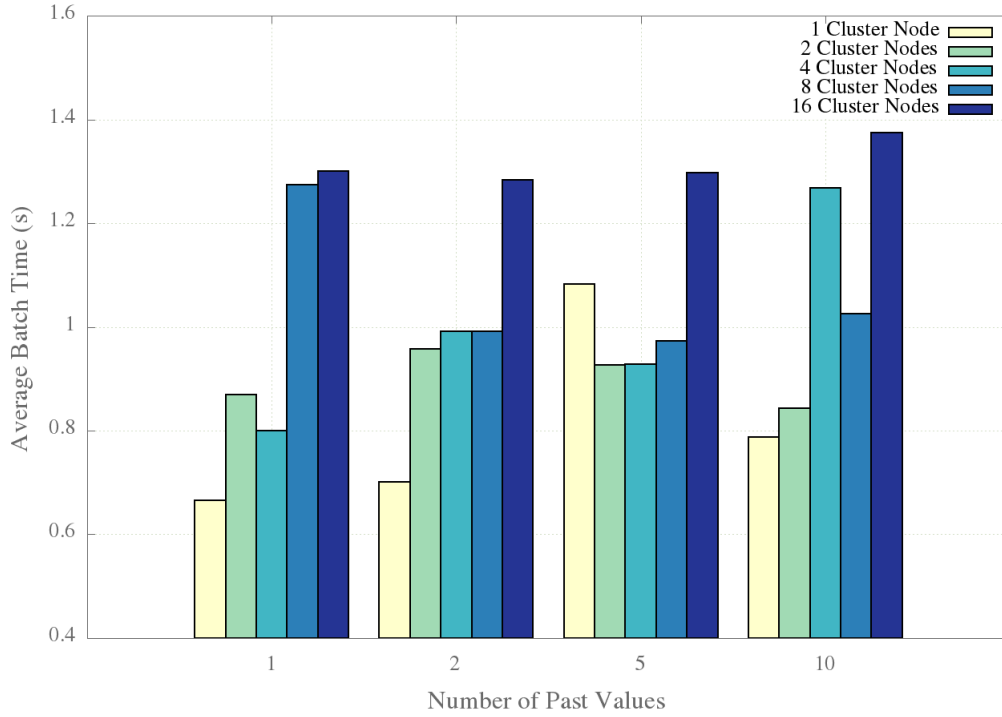


Figure 4.5: Temporal LISA - Average Duration for Different K's

## 4.4 Impact of Topology Density

To investigate a further detail of the calculation, namely the number of neighbours per node, the algorithm used for Section 4.2 was run against different topology types. Firstly, a sparse topology was used, with a low number of connections so that every node is connected with at least one other node. This topology is not connected, i.e. it forms a disconnected graph with a minimum degree of 1. Secondly, a connected topology was used, i.e. it forms a connected graph. Finally, a dense topology was used, in which all possible connections occur, i.e. corner nodes have a degree of 2, other border nodes have a degree of 3 and all other nodes have a degree of 4. For each of these topology types, topologies with 1600 nodes were generated and evaluated by using 16 cluster workers.

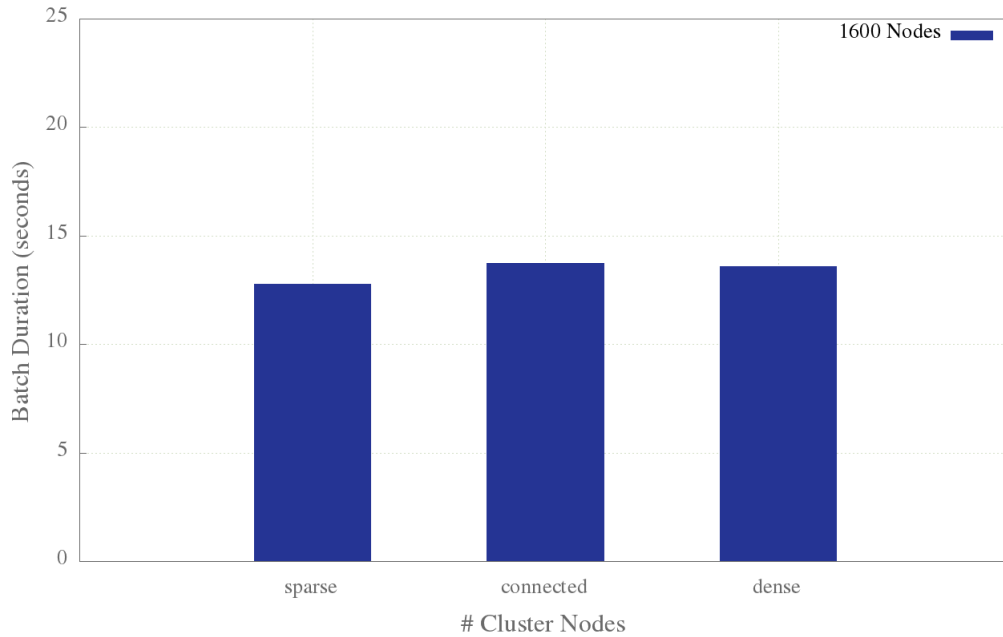


Figure 4.6: Average Calculation Duration for Different Topology Types

| Type      | Size | Number of Edges | Average Calculation Time |
|-----------|------|-----------------|--------------------------|
| Sparse    | 1600 | 1103            | 12.799 s                 |
| Connected | 1600 | 1599            | 13.736 s                 |
| Dense     | 1600 | 3120            | 13.596 s                 |

Table 4.4: Calculation Durations for Topology Types

Figure 4.6 depicts a comparison of the average calculation times for each topology with the according values given in Table 4.4. The histogram shows that the calculation is marginally faster with the sparse topology than with the connected topology. On the other hand, the performance of the calculation with the sparse topology is virtually equal to the performance with the connected topology. As each node has a maximum of four neighbours, however, this result is expected - the complexity of the calculation does not increase significantly with this low number of additional nodes (cf. Figure 2.1) and hence the performance with the different topologies used is expected to be similar. The measured differences in the performance are likely to result from noise such as network or CPU usage from other applications running in the cluster during the runs.

## 4.5 LISA with Monte Carlo Simulation

A further experiment conducted was to run the LISA algorithm in combination with a Monte Carlo Simulation to test for statistical significance, as described in sections 2.1.4 and 3.2.3. More precisely, both the naive and the improved approach were evaluated. As before, the standard topology was used. However, due to the longer calculation durations, the input rate and batch window had to be adapted to one value per minute and one minute, respectively. Furthermore, the run duration was set to 20 minutes, and three runs were executed for one, two, four, eight and 16 workers.

set lmargin 15 set rmargin 15 set tmargin 7 set bmargin 7

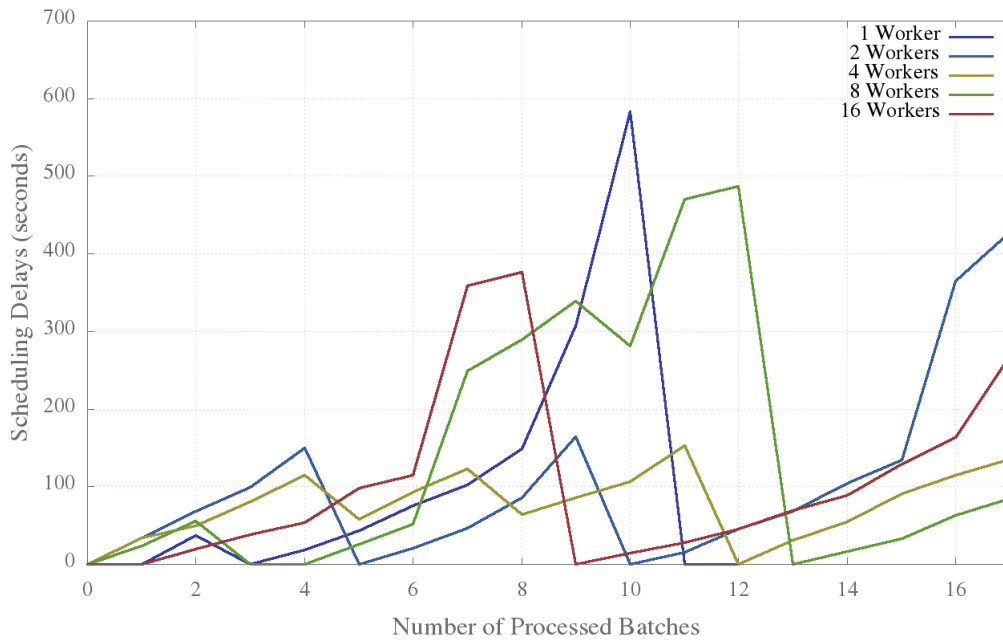


Figure 4.7: Scheduling Delays - Naive Approach

The first version of the algorithm proved to be rather inefficient, as visible from Figure 4.8. In fact, the speedup caused by increasing the number of workers from 1 to 16 was measured at approximately 1.3, which would not justify to run the calculation in a distributed way on a cluster. Furthermore, the average calculation duration of more than 100 seconds is generally very slow in a near-real-time context.

This algorithm nicely illustrates the effect of a too small batch interval configuration compared to the actual batch calculation durations. As shown in Table 4.5, in all cases the average batch duration remains well over the configured interval of 60 seconds. As a result, batches get increasingly delayed over time as shown in Figure 4.7. Furthermore,

at some point values which arrived during different intervals were processed in a single interval, which lead to incorrect results for the calculations. To address this issue, the batch duration would have had to be adjusted to over 100 seconds, which is not desirable for near-real time analytics of WDN sensor data. Hence, the algorithm was evaluated in detail in order to find improvements.

The job details, provided by *Spark* UI, indicated that one operation could be the underlying cause for the poor performance of the algorithm. This was confirmed by an analysis of the according log files. The `mapValues()` operation used to create values satisfying  $\frac{1}{N} \sum_{n=1}^N StdVal_{rk_n}$  for each key took on average 131.3 seconds for 1 worker, and 99.0 seconds for 16 workers. This accounts for the largest part of the calculation time. The according average calculation durations are depicted in Figure 4.8, representing the values listed in Table 4.5. Due to these results, the algorithm was adapted as discussed in Section 3.2.3.

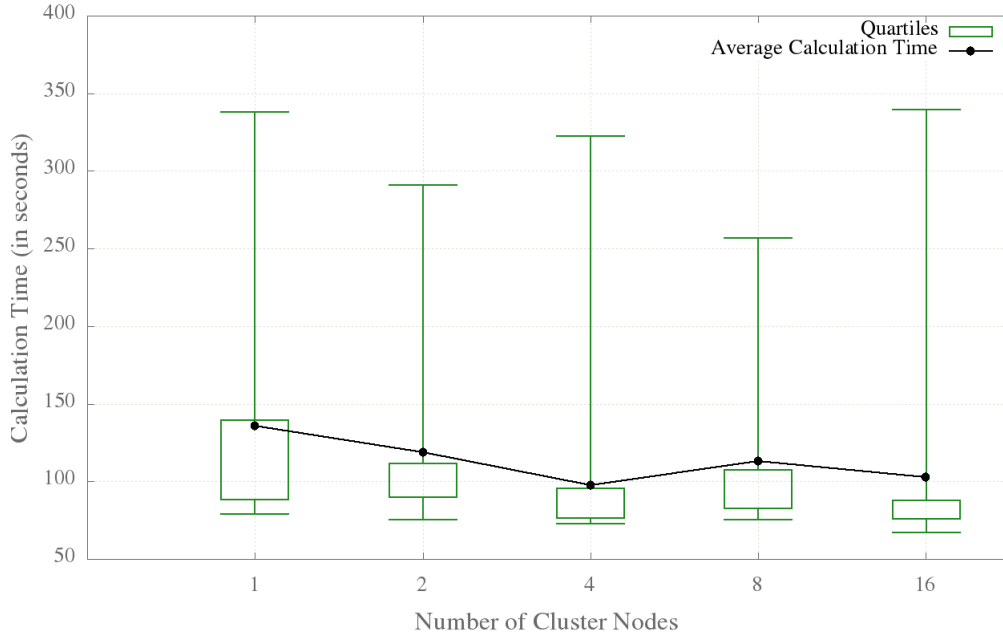


Figure 4.8: Monte Carlo Simulation Performance - Naive Approach

In contrast to the simple LISA calculations as well as to the naive approach presented above, the measured calculation durations in case of the improved Monte Carlo Simulation algorithm were more aligned with the expectation of a performance gain for an increasing number of workers (as depicted in Figure 4.9). As the averages in Table 4.5 show, a speedup of approximately 2.5 resulted from adding a second worker. With 16

workers, the speedup reached approximately 9 compared to one worker.

| Workers | Naive Approach | Improved Approach |
|---------|----------------|-------------------|
| 1       | 135.820 s      | 118.955 s         |
| 2       | 118.769 s      | 47.946 s          |
| 4       | 97.504 s       | 28.615 s          |
| 8       | 112.816 s      | 18.352 s          |
| 16      | 102.551 s      | 13.200 s          |

Table 4.5: Average Calculation Duration in Seconds

When comparing the graph of the improved algorithm (Figure 4.9) to the one of the naive algorithm (Figure 4.8), several differences emerge. Firstly, the performance improvement is quite drastic, as previously mentioned. Secondly, the average duration scales almost linearly with additional cluster workers. Thirdly, also the divergence of individual measurements decreases significantly, which means that calculation durations are favourable even in a worst case scenario when using 16 workers.

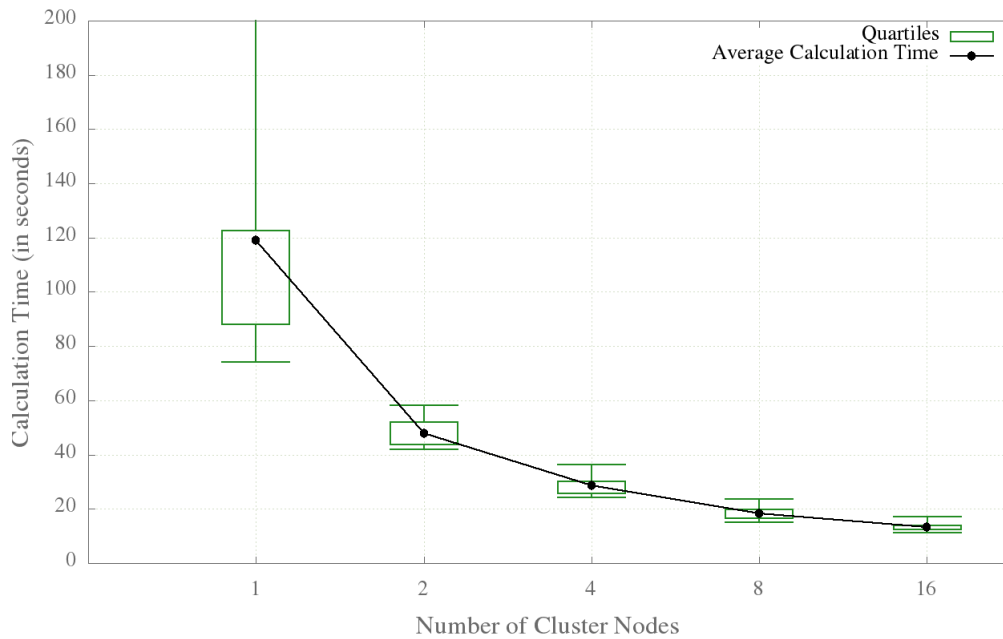


Figure 4.9: Improved Monte Carlo Simulation Performance



## 4.6 LISA with Temporal Association and Monte Carlo Simulation

As explained in Section 3.2.4, two variations of the Monte Carlo Simulation for temporal LISA statistics were implemented in the course of this thesis. Both variations were tested under similar conditions, namely using the standard topology and 16 cluster workers for all runs. However, the batch interval had to be increased to 200 seconds for the global selection algorithm (from 60 seconds for the local variation) in order to obtain valid results. Furthermore, both algorithms were tested for  $k \in 1, 2, 5$ . Higher values for  $k$  proved to be impracticable for the global selection due to high batch durations.

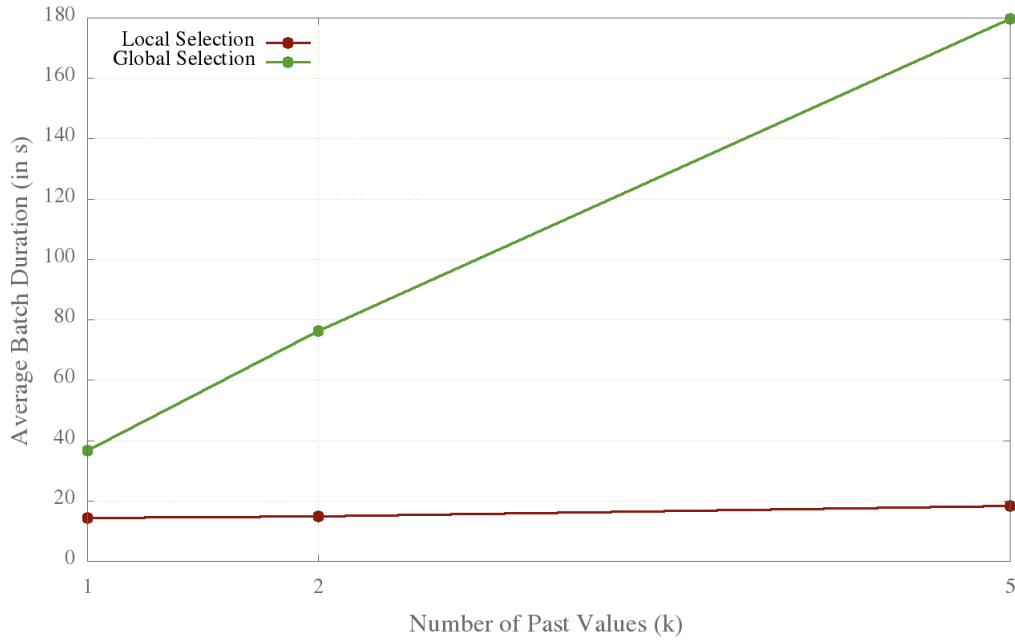


Figure 4.10: Temporal Monte Carlo - Algorithm Comparison

Figure 4.10 depicts the measured average batch durations for the different  $k$ s for both algorithm variations. The impact of considering additional past network states on the batch duration in the global selection variation is clearly visible. In fact, the increase of the calculations when increasing  $k$  is approximately linear, with ~36.67 seconds for  $k = 1$ , ~76.39 seconds for  $k = 2$  and ~179.78 seconds for  $k = 5$ . In case of the local selection variation, on the other hand, the impact of an increased value for  $k$  on the batch duration is much smaller. The average calculation duration increases from ~14.30 seconds for  $k = 1$  and ~14.88 seconds for  $k = 2$  to ~18.33 seconds for  $k = 5$ .

While the results for the global selection algorithm may seem to indicate bad overall performance, they are explicable and to be expected: for each additional network state, a full Monte Carlo Simulation has to be performed in which the complete global state has to be considered for each node value (cf. Section 4.5). This leads to a high level of wide dependencies in the computation graph of the algorithm, which in turn leads to long calculation durations.

# 5

## Conclusion

In chapters 3 and 4, both the implementation of LISA statistics on *Spark Streaming* and the performance of this implementation were discussed. It was shown that the implementation of algorithms for calculating LISA indicators and according statistical tests are feasible with *Spark Streaming*. Furthermore, the performance evaluation showed that the majority of these calculations can be run with reasonable performance in a near real-time manner.

The main conclusions which can be drawn from the implementation of this thesis is twofold. On the one hand, the high level API provided by *Spark Streaming* allowed for simple algorithms for LISA calculations. Even in the most complex case, namely the temporal Monte Carlo Simulation, the algorithm could be implemented on a moderate complexity level. On the other hand, recreating the WDN metering scenario described in [13] proved to be difficult mainly due to the characteristics of *Spark Streamings* data input API. This lead to several abstractions in the application architecture.

The performance evaluation for the LISA calculations have showed that *Spark Streaming* is well suited to handle the workload generated by a WDN even on a small cluster. The spatial LISA algorithm was run in various scenarios, namely with different numbers of nodes and different topology types in terms of the amount of connections. Furthermore, the performance impact of considering past network states was measured by running several configurations of the temporal LISA algorithm. In all cases, the calculation

duration per batch remained in a range which would easily allow for an ad-hoc analysis in a real-world WDN scenario.

However, one peculiarity was found when comparing different numbers of cluster workers. The results showed decreasing performance with additional cluster workers. This effect may result from the nature of LISA indicators as the calculation of both the standard deviation and the mean requires knowledge of the complete network state, which leads to unfavourable wide dependencies. Further work will be required to fully clarify this issue, as discussed in the subsequent section.

The evaluation of the statistical tests yielded more divergent results. In the spatial variation, reasonable delays were achieved with an algorithm adapted to the characteristics of *Spark*. The temporal variation of the algorithm, on the other hand, proved to perform poorly due to its dependency on the complete current and past network states. Reasonable performance could only be achieved with an adjustment proposed by [13].

## 5.1 Future Work

The problem covered by this thesis is very specific in terms of both technology (*Spark Streaming*) and of applicability to a real-world scenario (WDN). However, its scope had to be confined considerably. Consequently, there are many possibilities for improvements and further work. These include improvements of the algorithms and of the architecture, as well as the adaptation towards a realistic scenario.

As shown in Chapter 3, statistical algorithms are a central part of the implementation. As they were mainly developed as a proof of concept of using *Spark Streaming* for such calculations, they are in no way optimised for performance. Improvements could include a better adaptation to the characteristics of *Spark Streaming*, and to data-parallel calculations in general, as well as the exploitation of data locality.

In addition, while the implementation architecture is based on the scenario described by [13], many adjustments and abstractions had to be made. Clearly, an interesting complement to this thesis would be to integrate the implementation into this scenario with less adjustments and to test it in a real-world setting.

A crucial part of such an extension would be an adjustment of the data input architecture. Currently, sensor values are simulated with respect to the *Spark Streaming* batch interval as discussed in Section 3.1.2. Sensors deployed in WDNs, however, do not emit measurements in a synchronised fashion. In consequence, calculation results could be falsified if multiple values measured by a single sensor would be processed within one

batch. A possible solution to this issue could be the use of input values which are an average of sensor measurements over a certain time span as input values.

In conclusion, this thesis has shown that *Spark Streaming* is suitable for anomaly detection in the context of sensors deployed in a WDN, but there is still much room for improvement and extension.

# Bibliography

- [1] Running spark on yarn. "<http://spark.apache.org/docs/1.0.0/running-on-yarn.html#configuration>". "[Online; accessed 10-30-2014]".
- [2] Spark api. "<http://www.scala-lang.org/api/current/index.html#scala.util.Random>". "[Online; accessed 09-24-2014]".
- [3] Spark streaming programming guide. "<http://spark.apache.org/docs/latest/streaming-programming-guide.html>". "[Online; accessed 10-05-2014]".
- [4] The apache software foundation announces apache spark as a top-level project. "[https://blogs.apache.org/foundation/entry/the\\_apache\\_software\\_foundation\\_announces50](https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50)", 2014. "[Online; accessed 09-23-2014]".
- [5] Apache spark vs. apache storm. "<http://stackoverflow.com/a/24125900>", 2014. "[Online; accessed 10-26-2014]".
- [6] Data parallelism. "[http://en.wikipedia.org/wiki/Data\\_parallelism](http://en.wikipedia.org/wiki/Data_parallelism)", 2014. "[Online; accessed 10-26-2014]".
- [7] Samza - spark streaming. "<http://samza.incubator.apache.org/learn/documentation/0.7.0/comparisons/spark-streaming.html>", 2014. "[Online; accessed 11-06-2014]".
- [8] Storm documentation - concepts. "<https://storm.incubator.apache.org/documentation/Concepts.html>", 2014. "[Online; accessed 10-26-2014]".

- [9] Storm documentation - rationale. "<https://storm.incubator.apache.org/documentation/Rationale.html>", 2014. "[Online; accessed 10-25-2014]".
- [10] Storm documentation - tutorial. "<https://storm.incubator.apache.org/documentation/Tutorial.html>", 2014. "[Online; accessed 10-25-2014]".
- [11] Task parallelism. "[http://en.wikipedia.org/wiki/Task\\_parallelism](http://en.wikipedia.org/wiki/Task_parallelism)", 2014. "[Online; accessed 10-26-2014]".
- [12] Luc Anselin. Local indicators of spatial association – LISA. *Geographical Analysis*, 27(2):93–115, 1995.
- [13] D.E. Difallah, P. Cudre-Mauroux, and S.A McKenna. Scalable anomaly detection for smart city infrastructure networks. *Internet Computing, IEEE*, 17(6):39–47, Nov 2013.
- [14] Pierre Goovaerts and Geoffrey Jacquez. Accounting for regional background and population size in the detection of spatial clusters and outliers using geostatistical filtering and spatial neutral models: the case of lung cancer in long island, new york. *International Journal of Health Geographics*, 3(1):14, 2004.
- [15] Xinh Huynh. Storm vs. spark streaming: Side-by-side comparison. "<http://xinhstechblog.blogspot.ch/2014/06/storm-vs-spark-streaming-side-by-side.html>", June 2014. "[Online; accessed 11-06-2014]".
- [16] V. Kalavri and V. Vlassov. Mapreduce: Limitations, optimizations and open issues. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1031–1038, July 2013.
- [17] Vasiliki Kalavri and Vladimir Vlassov. Mapreduce: Limitations, optimizations and open issues. In *TrustCom/ISPA/IUCC*, pages 1031–1038. IEEE, 2013.
- [18] Simpal Kumar. Real time data analysis for water distribution network using storm. Master’s thesis, University of Fribourg, May 2014.
- [19] Zhiqiang Ma and Lin Gu. The limitation of mapreduce: A probing case and a lightweight solution. In *CLOUD COMPUTING 2010, The First International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 68–73, 2010.

- [20] Phokham Nonova. Hdfs blocks placement strategy. Master's thesis, University of Fribourg, October 2014.
- [21] 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.
- [22] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
- [23] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.





# Appendix

## **A.1 Complete Source Code**

The complete source code resulting from this thesis is available at the following URL:  
<https://github.com/snoooze03/SparkLisa>

# **Erklärung**

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: .....

Bachelor ☐      Master ☐      Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Ich gewähre hiermit Einsicht in diese Arbeit.

.....

Ort/Datum

.....

Unterschrift