FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Scalability for data stream processing frameworks

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Andrés Carrasco
Matrikelnummer 1126772

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Christoph Hochreiner

Wien, 30. Juni 2016

Andrés Carrasco	Stefan Schulte

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Scalability for data stream processing frameworks

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Andrés Carrasco
Registration Number 1126772

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.-Ing. Stefan Schulte
Assistance: Christoph Hochreiner

Vienna, 30th June, 2016

Andrés Carrasco	Stefan Schulte

Erklärung zur Verfassung der Arbeit

Andrés Carrasco
Sollingergasse 2/18, 1190 Wien, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Juni 2016

Andrés Carrasco

Abstract

Stream Processing is no new topic in the literature, but due to the increasing interest for processing unbounded data streams, the topic has resurfaced and has gained attention in the recent years. Part of this is due to the growing number of interconnected simple-purposed devices we call the Internet of Things. These devices often produce unbounded streams of data, which are impractical to store for them to be processed later. In order to practically process this data, Stream Processing Frameworks (SPFs) like Apache Storm, Apache Spark, and quite recently Kafka Streams appeared. The design of these SPFs requires developers to allocate computing resources for the requirements of their applications beforehand, usually in a distributed manner. However, when processing volatile data streams unusual spikes can occur. If the resources allocated by the developers prove to be less than required to handle these spikes, their application will suffer delays in processing until the spike wears off. Moreover, overprovisioning systems often leads to waste of capital, as paid-for computational power is not being utilized to its full potential.

Today there is an apparent need for obtaining information in real-time. If a SPF cannot handle unusual spikes of information, they are impractical for real-time processing. Existing approaches for handling such volatile data streams create a trade-off between accuracy and latency. Moreover, they are also constrained by the resources allocated when configuring the system. The Cloud Computing paradigm offers a solution for this problematic of finite resources. Through the Cloud Computing paradigm, elastic resources can be made available to SPFs with no apparent bound. This approach, however, brings its own challenges. Therefore, we selected and evaluated one of the most prominent SPFs. We propose a prototype extending Apache Storm for evaluating its performance by adding computational resources at run-time, utilizing OpenStack to add and remove Virtual Machines, depending on the needs of the topology. We implemented a simple global threshold algorithm, utilizing the number of available messages to process as a metric for scaling decisions. We concluded that this approach can have an impact on the application's throughput. However, great care needs to be taken in the designing of aggregation strategies, as wrongly designed strategies can create bottlenecks in the application. Potentially reducing substantially the performance of the application, as well as opening the possibility of cost-effective processing.

Contents

CHAPTER 1

Introduction

As the need to process large amounts of information arose, technologies appeared that were able to process such incredibly big amount of data like Apache Hadoop[1]. Apache Hadoop and similar frameworks, such as HPCC[2] and Hydra[3], rely on processing persisted data sets, revealing desired information from them. However, not all data processed today is persisted in a static manner. Transient data streams are becoming increasingly common, especially due to the emerging Internet of Things. Stream Processing Frameworks (SPFs) like Apache Storm[4] or Apache Spark[5] have gained a lot of traction in recent time for allowing the development of applications that can handle such data sets in a distributed and almost real-time manner.

A data stream is typically composed of sequential tuples made available over a period of time by a data source. These streams can be either constant or volatile. A constant data stream does not change its data rate, i.e., the amount of tuples exposed, over time. The most common source of constant data streams are sensors, e.g., temperature sensors which yield the measured temperature, depending solely on the refresh rate. A volatile data stream, opposed to constant data streams, does not maintain a constant data rate. For example, a traffic monitor detects more vehicles during rush hour than in the middle of the night [LTDH+14].

Processing constant data streams do not present a big challenge, as the computational resource requirements do not vary over time and only need to be configured once. However, volatile data streams are challenging, since the SPF must adapt continuously to the changing rates for achieving real-time processing [SeZ05]. A solution would be to increase and decrease the computational power of the underlying system. This is referred to

[1] http://hadoop.apache.org/
[2] https://hpccsystems.com/
[3] http://findwise.github.io/Hydra/
[4] http://storm.apache.org/
[5] http://spark.apache.org/

as *vertical scaling*, where the action of adding extra resources to an existing host is called *scale up*. The contrary action of removing resources is to *scale down*. Whenever a system cannot cope with the processing tasks, the system is said to be *underprovisioned*. Whereas a system that has too many unused resources while processing is said to be *overprovisioned*. The dynamic adaptation of computational resources for coping with underprovisioning or overprovisioning is regarded as *elasticity* [DGST11]. This helps avoid overprovisioning and underprovisioning scenarios, guaranteeing a high QoS (Quality of Service) [AFG+10].

The problematic around over- and underprovisioning systems is mainly economical. Developers had to try and predict, mainly by guessing, the required computational resources of their application. As a result, the computational systems resources often presented overprovisioned or underprovisioned scenarios. Deploying overprovisioned systems, translates into a waste of capital, as paid computational power is not utilized to its complete capacity. Whereas deploying underprovisioned systems, can translate into a loss of potential customers when unusual and unexpected interest peaks occur [AFG+10].

To create cost-effective provisioning of resources, the Cloud Computing paradigm was developed. Through Cloud Computing a pool of computational resources is made available to consumers, allowing them to automatically allocate them on-demand [MG11]. The way a Cloud Computing service is provided can be at different levels of abstraction. The lowest level of abstraction is the *Infrastructure as a Service (IaaS)*, in which typically processing power, storage or networks are directly provided, allowing the execution of arbitrary software, and operating systems. The *Platform as a Service (PaaS)* works on top of the latter, providing the consumer with applications, libraries, services and tools to develop and deploy applications that run on cloud resources. Lastly, in the topmost abstraction, lies the *Software as a Service (SaaS)*, in which an application that utilizes cloud resources, is made available through a client or a programming interface [AFG+09]. Typically in a Cloud Computing IaaS environment, virtual computing resources are created or destroyed on the fly in the form of virtual machines, where the action of creating resources is referred to as *scale out*, while the destruction of resources is called *scale in*. This is referred to as *horizontal scaling*. Heinze et al. [HPJF14] point out that efficiently scaling in and out, is a major challenge for modern SPFs, as through them a stable latency and an optimal system load can be achieved, having thus a proper QoS.

Hummer et al. [HSD13] present the Cloud Computing paradigm as an ideal infrastructure for Elastic Stream Processing, due to their resource elasticity and pay per use model. They identified four core challenges an Elastic SPF must fulfill in order to utilize cloud resources effectively. First, the framework must have an optimized integration with specific cloud environments, as well as adaptation algorithms to exploit available resources. Second, the framework must be able to offer stream processing as a service, demanding as a side effect the security and privacy of consumers of such service. Third, the framework must have suitable monitoring infrastructure to ensure customizable service-level agreements are not violated. Fourth and last, the framework must also be able to reduce latency, by providing context- and locality-aware data delivery. Through horizontal scaling, the first

	Borealis	Storm	ChronoStream	VISP
Cloud Resources Exploitation		(✔)	✔	✔
Stream Processing as a service			✔	✔
Monitoring Infrastructure				✔
Context & Locality-aware data delivery				✔

Table 1.1: Challenges for ESP on Hummer et al. [HSD13]

challenge of Hummer et al. [HSD13] can be solved, as computing nodes can be added to a distributed SPF on the fly, achieving thus resource elasticity [DGST11].

In Table 2.1 a comparison of some of the discussed distributed SPFs can be found, in terms of the latter described challenges. The recently published VIenna ecosystem for elastic Stream Processing (VISP) has presented solutions for all of the four challenges [HVWD16]. ChronoStream [WT15] has effectively tackled the exploitation of cloud resources, providing a multi-tenant environment for stream processing. Even though ChronoStream achieved elastic stateful stream processing, no monitoring infrastructure was provided as well as support for locality-aware data delivery. Lastly, Storm has the possibility of manually supporting resource elasticity through the `rebalance` command, however, it fails to automatically exploit cloud resources [CNL16].

Within the scope of this thesis, we identify different approaches for coping with volatile data streams, as well as their challenges. We have identified three different approaches utilized in different SPFs. The first identified approach, called *load shedding*, is utilized in Aurora[6] and Borealis[7]. In this approach, incoming tuples are dropped whenever the system's capacity is overloaded. In the second identified approach, the tuples are assigned to and balanced between different nodes in a distributed computing cluster. This approach is referred to as *load balancing*. In the third approach, the *auto-scaling* approach, computing nodes are made available at run-time, for increasing or decreasing computing resources, depending on the demand. Whenever the system's capacity is overloaded, nodes are added for increasing the system's total distributed capacity. After the status of the system returns to and underloaded state, the recently created nodes are then released. The challenges the latter present are: the ability for responding within seconds to fluctuations in the workload, the prohibition of modeling fixed workload models, and lastly, the system must be able to monitor each of the nodes capacity [HPJF14].

To the best of our knowledge, established SPFs do not include an auto-scaling strategy. However, Apache Storm claims the resources utilized by the framework can be dynamically allocated[8], opening thus the possibility of scaling automatically Storm *topologies*. Topologies in the context of Storm can be described as a directed graph, in which each of the nodes contains processing logic. This graph describes the way components are

[6] http://cs.brown.edu/research/aurora/
[7] http://cs.brown.edu/research/borealis/public/
[8] http://storm.apache.org/releases/1.0.1/Understanding-the-parallelism-of-a-Storm-topology.html

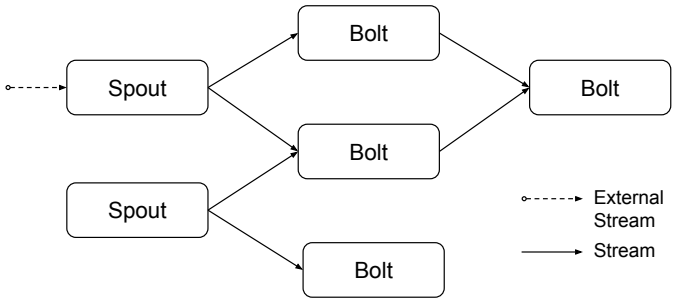


Figure 1.1: A Storm Topology Example

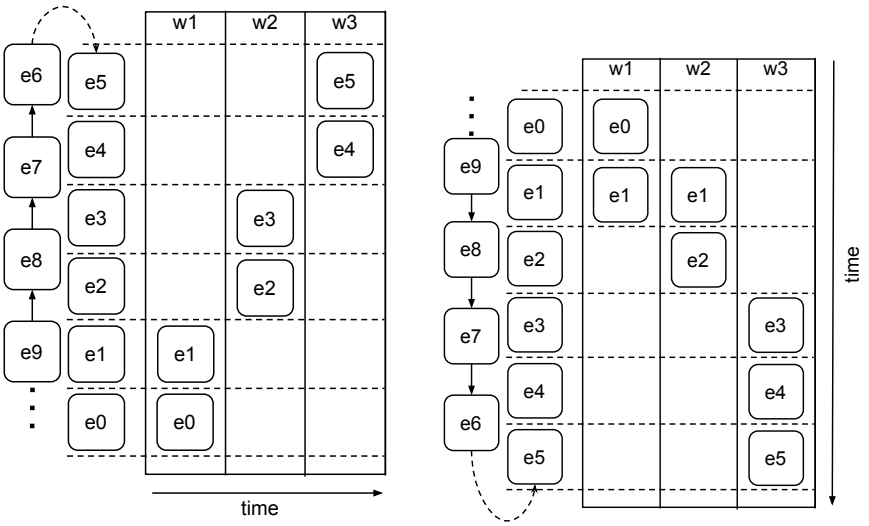
interconnected or arranged, i.e., which nodes are subscribed to the emissions of other nodes. An example of such arrangement can be found in Figure 1.1. Each of these nodes can be either a *Spout* or a *Bolt*. Spouts can generate tuples on their own, or receive them from external sources, subsequently emitting them to subscribed Bolts. A Bolt typically is subscribed to one or more spouts, and possibly to other bolts. Through this subscription, a number of input tuples are fed for it to process and possibly emit tuples to further bolts. An arrangement of this kind is deployed when starting a Storm processing job. A deployed arrangement cannot be modified at run-time. To modify it, the topology must be redeployed, making it inflexible for evolving requirements [HVSD16].

The aim of this thesis is to identify and discuss the methods utilized to cope with volatile data streams, as well as implement and evaluate an auto-scaling prototype extending Apache Storm's functionalities. Furthermore, in Chapter 2 a survey on Stream Processing is presented, with the goal of analyzing and describing current research efforts in this field. In Chapter 3 the research challenges of this thesis are discussed, as well as a brief overview of identified open issues. Additionally, a prototype for a scalable SPF with Apache Storm is presented in Chapter 4. Moreover, in Chapter 5 a discussion of the evaluation performed on the prototype is presented. Finally, our conclusions and findings are outlined in Chapter 6.

CHAPTER 2

Related Work

2.1 Basic Concepts of Stream Processing



(a) A Tumbling Window: whenever a window has ended, a new one is created

(b) A Sliding Window: every specified time, a new window is created

Figure 2.1: Tumbling and Sliding Windows

Generally, the terms *Stream Processing*, *Stream Computing* and *Complex event processing (CEP)*, are used interchangeably to refer to the processing of data streams as input [HSD13]. In Stream Processing, the basic data unit is referred to as an *event*. An event has a type, a timestamp, and further specific data. In the literature, an event can also be referred to as a tuple, stream data, object, or data item. In the terminology of

Storm, the word *tuple* is used as a synonym of event. Whenever aggregating, filtering or similar actions are taken on a set of events, the resulting event is called a *complex event*. Components that emit events are *event sources*, while those that consume them are *event sinks*. These components are called Spouts and Bolts in Storm, respectively. A linear sequence of events with no apparent end, usually ordered by time, are referred to as *event streams* or *data streams*. These unbounded event streams can be sometimes partitioned to create finite sets referred to as *windows*. Whenever the partition size considers only one event at a time, is referred to as a *single event window*. Other types of windows include tumbling windows, and sliding windows, among others. In the *tumbling windows*, shown in Figure 2.1a, the incoming events are aggregated together in a window and made available for processing. The aggregation is made by time, in which the incoming events inside a timespan are grouped together. However, this aggregation can also be made by means of an event count. Whenever the desired count of events is reached, the window is made available for processing. In *sliding windows*, shown in Figure 2.1b, the events are made available through a window with a certain time length, in which all of the events received within a timespan are inside the window. After a given interval, the window discards the first received events, for later to include the next received events outside the window, hence it slides. The number of events to discard and include depends on the length of the interval. Other window types, such as landmark windows, are discussed by Patroumpas and Sellis [PS06].

The *Event Processing Agents (EPAs)* software module takes care of consuming, processing and emitting new events. In other words, EPAs ensure the operation of Spouts and Bolts. Storm named this component as the Supervisor. *Event Processing Networks (EPNs)* are directed graphs describing the way event sources and event sinks are interconnected or arranged. In Storm terminology, EPNs are the aforementioned *Topology*. For the sake of consistency, the terminology of Storm will be used henceforth in this thesis.

2.2 Early Work on SPFs

Early efforts to create systems capable of dealing with data streams begin with Aurora. Aurora's successor, Borealis was later implemented to extend Aurora's monolithic architecture, to create the first SPF with a distributed architecture [AAB+05]. Borealis achieved a higher performance through parallelization and increased fault tolerance, in contrast to its predecessor [TAe+06]. Both Aurora and Borealis have *load shedding* mechanisms, to help them cope with changing data rates. Load shedding mechanisms remove tuples before being processed for reducing the current processing load. This mechanism triggers when the system cannot handle the number of input tuples. As a consequence, Aurora's random load shedding algorithms lead to a utility loss in tuples [BBC+04]. To avoid this loss of utility, Borealis implemented a priority scheduling queue, which allowed the processing of tuples, not in time order, but rather in priority order. Thus, enabling the semantic dropping of tuples with low utility when the system was overloaded [AAB+05]. However, load shedding creates a trade-off between low-latency and accuracy. Due to the loss of information in the dropped tuples, this approach has a negative impact on the

quality of the results [TZ06]. Load shedding is an optimization problem, dealing with the challenges of when, where and how much load to shed [TeZ+03]. Utilizing load coefficient, calculated using the computational costs of stream operators, as well as their ratio of output and input rate or *selectivity*, the first challenge can be solved. The other two challenges, the where and how much, can be solved using a *load shedding roadmap*, determining the processor's *cycle savings coefficient* at different points in the network [HSD13]. Due to time constraints, load schedulers cannot delay the computation of optimal plans, therefore potential plans are built in advance with offline algorithms [TeZ07]. Whenever a high difference between average workload and actual workload occurs, a system that must discard most of the tuples when accuracy is needed is suboptimal [KKP11].

2.3 Real-time processing on SPFs

Stonebraker et. al. [SeZ05], analyzed classical Database Management Systems (DBMS), rule engines, and SPFs, for determining which technology is best suited for processing data streams in real-time. Due to SPF's innate capability to address stream processing requirements, they were better suited for supporting real-time processing of streams. In order to process data in real-time, Stonebraker et. al., outlined eight requirements for SPFs:

- **Rule 1: Keep the Data Moving.** The first requirement is to process data without the need store them for conducting operations, as well as actively process them i.e., avoiding polling conditions. This is due to the fact that disk operations add an unnecessarily big latency on processing. Whereas polling for changes, possibly adds a delay between the actual occurring of the event and its processing.
- **Rule 2: Query using SQL on Streams.** The use of a high-level query language with built-in primitives and operators, suitable for stream processing, e.g., merge, join, or aggregate. The reasoning behind this requirement is based on the fact that streaming data never ends, e.g., a standard SQL `SELECT` would not know when to stop, given an unbounded table of elements. For that reason, *window* constructs are deployed, for defining a scope of the data to process.
- **Rule 3: Handle Stream Imperfections (Delayed, Missing, and Out-of-Order Data).** In contrast to conventional databases, data in a real-time system can be late, missing or out-of-order. When doing calculations, a timeout must be always present, especially on blocking calculations where more data is needed in order to process further. This is due to the possibility, of that waited-for data might never appear, thus blocking the system indefinitely, hindering its ability to continue. Moreover, data can also come out-of-order. The SPF must be able to generate predictable results regardless of the order.
- **Rule 4: Generate Predictable Outcomes.** The SPF must also ensure their results are predictable and repeatable. This requirement has also to do with the

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Amazon Kinesis	✔		✔	✔		✔	✔	✔
Cloud Dataflow	✔	✔	✔	✔	✔	✔	✔	✔
Apache Spark	✔	✔	✔	✔	✔	✔	✔	✔
Apache Storm	✔	✔	✔	✔		✔	✔	✔

Table 2.1: Comparison of SPF on Stonebraker et al. [SeZ05]

previous. Even when data comes completely out-of-order, in order to remain predictable, the SPF must be able to yield the same outcome.

- **Rule 5: Integrate Stored and Streaming Data.** The fifth requirement is to be able to seamlessly handle historical data as well as live streaming data. Often comparing historical and present data is needed in stream processing applications. Therefore, the SPF must be able to seamlessly switch between both historical and live streaming data.

- **Rule 6: Guarantee Data Safety and Availability.** A SPF must also ensure the availability of the system and integrity of the data. In the case of failure, the SPF must be able to continue processing in backup systems. Especially when the processing of the data is critical for the consumer.

- **Rule 7: Partition and Scale Applications Automatically.** The SPF must be capable of distributing processing tasks between multiple processors and machines, in other words, it must be able to scale. This has as a consequence a better performance and overall optimal price for computation, as resources are better exploited.

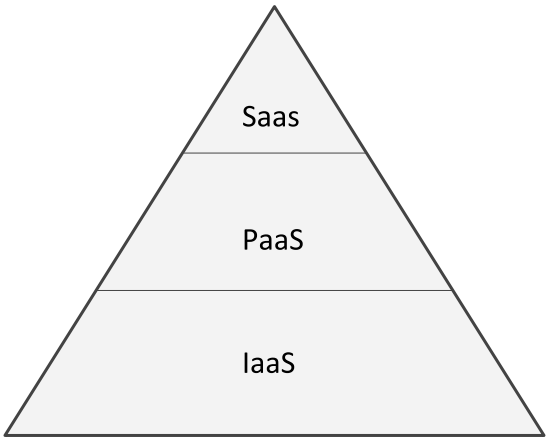
- **Rule 8: Process and Scale Applications Automatically.** Lastly, the SPF must have minimal overhead in its execution and be optimized, so that it can respond in real-time. Without an optimal set of instructions, an unintended overhead can build up and increase the execution time, reducing the overall performance. Therefore, all of the components must be designed for high-performance, so that the execution overhead can be maintained to a minimum.

This 8 rules, were outlined in the previous decade. However, they have transcended until today, as a base metric for guaranteeing reliable real-time processing. Therefore we have compiled a table (See Table 2.1), to the best of our knowledge, of how some of the modern SPFs measure up against these requirements. Amazon Kinesis[1] utilizes a data record, which persists incoming data for later to be processed in-order by Kinesis Stream Applications. However, their API does not provide windowing, therefore violating the Rule No. 2. Moreover, both Kinesis and Storm fail to meet Rule No. 5, as they do

[1] https://aws.amazon.com/kinesis/streams/

not provide a seamless way to integrate stored data. In contrast to Kinesis and Storm, Google's Cloud Dataflow[2] and Apache Spark seem to fulfill all of the requirements.

2.4 Cloud Computing



Adopted from http://filiph.net/

Figure 2.2: Cloud Computing Services

The term *Cloud Computing* refers to services made available through the Internet, as well as the underlying hardware and software in data centers. Typically, when referring to a *Cloud*, the underlying hardware, and software in data centers is meant. Whereas the services made available, are named as *Software as a Service (SaaS)*. Under the SaaS, another two layers are working to deliver services to consumers. Namely the *Platform as a Service (PaaS)* and *Infrastructure as a Service (IaaS)*. A representation of this layering can be seen in Figure 2.2. As the name suggests, SaaS makes software and services available to end consumers. In the PaaS applications, libraries, services and tools to develop and deploy applications that run on cloud resources are made available to consumers. Lastly, in the IaaS, processing power, storage, or networks are typically provided, allowing the execution of arbitrary software, and operating systems [AFG+09].

In an effort to standardize this paradigm, Mell and Grance [MG11] defined and identified the characteristics that the Cloud Computing paradigm must exhibit:

On-demand self-service the user can automatically adjust the computing capabilities, such as network storage, server time, networks, servers, etc., without the need of a physical interaction with the service provider.

[2] https://cloud.google.com/dataflow/

Broad network access the capabilities must be accessible through standard methods, e.g., the Internet.

Resource pooling the provided computational resources are pooled or shared among multiple users, assigning dynamically virtual and physical resources according to the users demand.

Rapid elasticity the capabilities can be created and released, to scale depending on demand without an apparent bound.

Measured service the resources utilized can be monitored, quantified, and reported for creating a pay-per-use basis. These resources can also be controlled and optimized in respect to the latter.

When deploying applications, developers had to predict and allocate computational resources for their services. However, these predictions often resulted in overprovisioning, as the popularity of the service might have been overestimated, or in underprovisioning, if the popularity was underestimated. Deploying overprovisioned systems, translates into a waste of capital, as paid computational power is not utilized to its complete capacity. Whereas deploying underprovisioned systems, can translate into a loss of potential customers when unusual interest peaks occur [AFG+10].

As a solution to this problematic and to effectively provisioning of systems, the Cloud Computing paradigm appeared. Through Cloud Computing a pool of computational resources is made available to consumers, allowing them to automatically allocate them on-demand [MG11]. Whenever there is an increase in demand, more resources can be leased to the consumers. On the contrary, whenever the demand is reduced, some of the allocated resources can be released. This dynamic allocation and release of resources are referred to as *scaling*. Scaling can be split into two categories: *vertical scaling*, and *horizontal scaling*. In vertical scaling, the action of adding extra resources to an existing host is called *scale up*. The contrary action of removing resources is to *scale down*. In horizontal scaling, typically extra hosts are created or destroyed on demand, where the action of creating a host is referred to as *scale out*, while the destruction of a host is called *scale in*. Typically in Cloud Computing resources are virtualized and made available to the consumer via Virtual Machines (VM). Virtual Machines are emulations of a computer system, running atop a pool of resources. Whenever horizontal scaling is needed, a cloud provider can add or remove VMs depending on the demand [HBS+16]. The actions of scaling in or out, together with the possibility of configuring when the actions will happen, as well as health-checking the running instances, is referred to as *auto-scaling* [ALHL14]. Lastly, the pay-as-you-go model of cloud computing, allows the consumer to expend only in what actually was utilized, translating into an economic benefit for the customer [AFG+10].

2.5 Elastic Stream Processing

Lately, continuous data streams have become increasingly common, especially with the popularization of the Internet of Things. Hochreiner et. al. [HSDL15], raise the need for SPFs to be elastic in three dimensions with regard to their processing capabilities to maintain service-level agreements (SLAs). The first dimension is the *quality elasticity*, dealing with the quality of responses, or their response time. The second dimension is the *resource elasticity*, which deals with the adaptation at runtime of computational resources to cope with the underlying system's load. In the third dimension, the *data elasticity* dimension, the data being distributed and made available to the SPF is dealt.

In respect to resource elasticity, some work has been done to better utilize multi-cored architectures of computational nodes by employing runtime elasticity, finding the optimal resource utilization within the node [SAG+09]. Other efforts on applying *data parallelization*, i.e., distributing tuple processing throughout a distributed system, dynamically utilizing the available computational nodes effectively [SAG+09].

With the introduction of the Cloud Computing paradigm, computational nodes could be made available to running systems, thus achieving resource elasticity [AFG+10]. As the Cloud Computing paradigm became more established, SPFs began integrating distributed computing resources to achieve run-time processing, instead of overprovisioning, resulting in a cost-efficient data stream processing system [IS11].

As stream processing received more attention, from the realm of Open Source software, several other SPFs started appearing e.g., Apache Storm[3], Apache Spark[4], and quite recently Kafka Streams[5], among others. Hochreiner et. al. point out, that these SPFs only support fixed stream processing topologies, i.e., they need to be resubmitted if the configuration or the topology itself was modified, making them unfeasible for the volatile Internet of Things. In addition to the open source SPFs mentioned before, proprietary solutions also exist, such as Amazon IoT[6] and Google Cloud Dataflow[7]. They are capable of running multiple parallel topologies and distribute accordingly over a large pool of resources [VPK+15].

Typically, deployed stream processing applications feed on data sources that, through the Internet, are located at various points around the globe. The distance between the processing cluster and the source can impose a burden on the communication infrastructure, potentially adding latency and thus hindering its efficiency. This challenge is known as the *Placement Problem* and is known to be NP-hard [CGLPN16]. To solve the placement problem, a heuristic approach is needed. Current different approaches aim to heuristically reduce the end-to-end latency, the inter-node traffic, and the network usage. Through these heuristics, the host that should execute an operator, i.e., a Bolt or

[3] http://storm.apache.org/
[4] http://spark.apache.org/
[5] http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple
[6] https://aws.amazon.com/iot/
[7] https://cloud.google.com/dataflow/

Spout in Storm's terms, is selected [CGLPN15]. Cardellini et. al. [CGLPN16] evaluated each of this heuristics with Storm compared against their solutions computed by CPLEX[8], resulting in promising results for small scaled problems, however degrading with larger scales.

2.6 Auto-Scaling

One of the challenges of scaling data SPFs is to decide when to add or remove computing nodes. Heinze et al. [HPJF14], proposed three basic auto-scaling techniques in the context of elastic stream processing, using local and global threshold-based approaches, as well as a reinforcement learning technique. The scaling decision was taken depending on the host's capacity utilization. The reinforcement learnings showed to be the most resilient and adaptive solution. Further work needs to be done, to include more parameters than the ones proposed in the latter work. Scaling decisions should also be taken in respect to quality elasticity, e.g. reducing the cost-effectivity to increase the application's throughput. Other proposed solutions include utilizing an upper threshold for all of the load in the nodes [GJPPM+12], while another solution includes both an upper and lower threshold for each individual node [CFMKP13].

2.7 Run-time Self-Configuration

Considering the role SPFs currently play in the Internet of Things (IoT), the challenges it presents must also be addressed. Due to the volatility of devices in the IoT, SPFs must be able to support self-configuration and the addition of operators [HVSD16]. In this context, the typical centralized approach is not practical [MSDPC12]. Moreover, most of the SPFs do not support modifications of topologies at run-time, typically topologies need to be stopped, modified and redeployed, potentially leading to loss of time and utility.

Most of the research done so far has focused on solving some of the problems. Hochreiner et. al. [HVWD16], proposed VISP, in which the inflexibility of SPFs in terms of configuration is tackled, as well as easing the creation of topologies.

2.8 Stateful Stream Processing

In Stream Processing, streams can be either *stateless* or *stateful*. In stateless streams, there is no relation between tuples, making the order of their processing irrelevant. In contrast to this, stateful streams do have a relation between tuples, requiring complex operators like windows [HSD13]. Stateful streams pose a challenge for SPFs, as states must be migrated when a computing task is assigned to another computing node [CNL16]. This migration of states can pose a delay in processing due to I/O network operations [WT15].

[8] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Wu et. al. [WT15], propose a slice computation and reconstruction approach, in which states are divided into computational slices and distributed, achieving an abstraction from the system's OS concurrency. This approach yielded a notable improvement with respect to state migrations. Cardellini et. al. [CNL16], propose a *pause-and-resume* approach, in which the state is extracted from the old instance, and it is replayed in within the new. Another approach utilized is the *parallel track*, in which the old task and the new task at the same time until a sync is achieved, hence removing the need for the old task, allowing its removal.

CHAPTER 3

Research Challenges and Open Issues

Established SPFs have been widely adopted, due to their excellence, and versatility in processing data streams. However, with the ever-increasing needs of todays information-centric society, SPFs need to continuously evolve, to meet the modern demands.

The challenges for SPFs are vast, many of them have been tackled, however some still remain. The aim and scope of this thesis is presented in the following section, starting with the research challenges this thesis presents in Section 3.1, succeeded by the identified open issues that were identified in Section 3.2.

3.1 Research Challenges

This thesis deals with the most noticeable missing requirement for SPFs: the possibility to scale automatically. Not only does scaling allow cost-effective processing, but also gives the possibility of scalable real-time processing. For this reason, within this thesis, we will focus on two research questions;

1. **Which approaches are possible to deal with a varying amount of streaming data?**

   The core functionality of SPFs is the ability to process streaming data effectively. However, whenever the amount of streaming data variates, especially when it increases, the system must be able to continue processing all of this data in an effective and accurate way. Should the system not be able to guarantee this, there is a big loss of utility. Therefore, a SPF must be able to deal with varying rates of streaming data, and thus discussed within this thesis.

2. **How are state of the art stream processing systems capable of dealing with varying amounts of streaming data?**

   There are certainly multiple strategies, such as load shedding and load balancing, however, these approaches are usually deployed with static computing resources. In other words, they have an inherent limit on their processing capabilities. Therefore, the possibility of using apparently limitless cloud resources must be researched on existing established SPFs, and whether some currently utilize these elastic resources.

3.2 Open Issues

Throughout the research performed in the writing of this thesis, some open issues were identified and are presented here as questions, together with brief general descriptions. The descriptions are meant to capture the importance of their future research. These questions are, however, not discussed further within this thesis.

Auto-Scaling of Stream Processing Frameworks

Auto-scaling itself opens a variety of possibilities for processing bigger data rates, faster and in a reliable way. Even though to the best of our knowledge, established Open Source SPFs do not support scaling at runtime, they provide re-balancing schemes that allow manually changing of parallelism and request a new load balancing calculation. Moreover, choosing when to scale is also a challenge alone, as scaling decisions make a notable difference in resource usage optimality. Therefore, some questions raised remain unanswered within this topic:

- Is it possible to provide an auto-scaling PaaS to scale arbitrary stream processing frameworks?
- Is it better to have an integrated auto-scaling scheme within the SPF, or have an external service that provides auto-scaling?
- Which auto-scaling policy provide optimal scaling decisions?

Stateful Stream Processing

Whenever scaling, stateless streams can continue processing without regard to states, as they are irrelevant for the output. However, scaling stateful streams cannot ignore their state if they wish to remain accurate. Therefore, states must be migrated, replayed or synced in order to continue accurately. Some literature was identified, dealing with the issue of scaling and states. However, some open questions still were identified:

- How can stateful stream processing programming can be seamlessly be provided to developers?

- Can stateful stream processing have a performance impact on multi-tenant systems?
- Would scaling up instead of out save time on costly state migrations?

CHAPTER 4

Approach for Auto-scaling

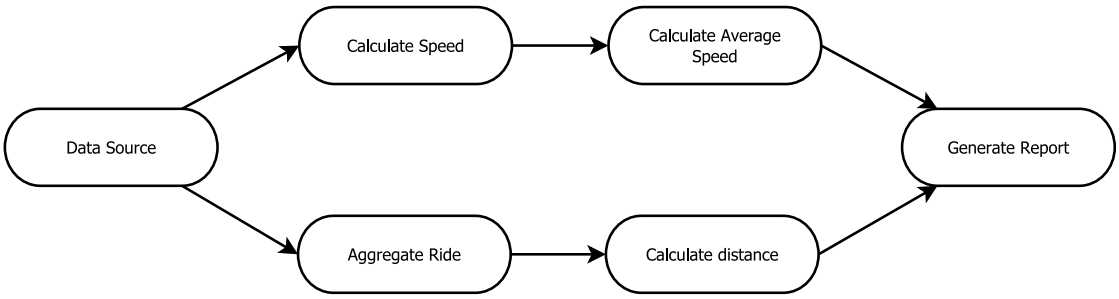


Figure 4.1: Deployed Topology

In order to evaluate the scaling capabilities of Apache Storm, a simple threshold algorithm was selected. Apache Storm will be deployed in an OpenStack[1] cluster, where the algorithm will scale in or out depending on the number of messages in a queue. As a queuing system, a RabbitMQ[2] server will be deployed in an independent node. Furthermore, we implemented a tool that will fill the RabbitMQ queue at a mostly constant rate. The threshold algorithm will scale in or out depending on the number of messages waiting in the queue. Lastly, a simplification of the deployed topology can be seen in Figure 4.1. A Storm specific topology will be presented as well in Section 4.4.

In the following sections, we will present our implemented prototype. In Section 4.1 the utilized test dataset will be presented. Afterward, in Section 4.2 the proposed scaling algorithm is presented. Subsequently, in Section 4.3 all of the utilized components will be discussed. The specifics for Apache Storms regarding scalability are discussed in Section 4.4. Lastly, the way our prototype was evaluated can be found in Section 4.5.

[1] https://www.openstack.org/
[2] http://www.rabbitmq.com/

4.1 Test Data

We selected a dataset, containing GPS trajectories of around ten thousand taxis in Beijing, totaling around 15 million tuples [Zhe11]. This data was collected between Feb. 2 to Feb. 8, 2008, totaling the distance traveled by all of the taxis combined up to 9 million kilometers. This data is organized in files, each file contains the tuples of a given taxi identified by a unique number. In order to reduce the total test time, the data of only 24 taxis selected, reducing the number of tuples to around 2.8 million.

Each tuple contains the following comma-separated fields:

```
(taxi id, date time, longitude, latitude)
```

The data is being persisted in a PostgreSQL[3] database in an independent computing node alongside the previously mentioned RabbitMQ server. During the processing, the data will be selected and ordered by the *date time* field, and sent to the RabbitMQ server, simulating the concurrency of such data, i.e., the way the server would get the data of multiple taxis in realistic scenarios. A Sample of the data can be found in the Table 4.1 below.

taxi id	date time	latitude	longitude
3579	2008-02-02 13:30:44	39.9035	116.40048
3015	2008-02-02 13:30:44	39.89171	116.41036
6275	2008-02-02 13:30:44	39.90484	116.36838
6275	2008-02-02 13:30:44	39.90484	116.36838
3579	2008-02-02 13:30:45	39.90341	116.40049
3015	2008-02-02 13:30:45	39.8917	116.41028
6275	2008-02-02 13:30:45	39.90484	116.36838
6275	2008-02-02 13:30:45	39.90484	116.36838
3579	2008-02-02 13:30:46	39.90334	116.40049
3015	2008-02-02 13:30:46	39.8917	116.41022

Table 4.1: A sample of the selected dataset ordered by date time.

4.2 Scaling Algorithm

For our implementation of an automatic scaling solution for Apache Storm, we selected a simple Threshold Algorithm. Our goal of utilizing such algorithm is to focus on Apache Storm's capabilities in adapting with different numbers of computing nodes at run-time, rather than evaluating the scaling algorithm's performance.

[3] https://www.postgresql.org/

The Auto-Scaling algorithm 4.1 consists of two elements: a *monitored resource*, and a *threshold*. The decision of scaling in or out is based on two specified parameters in the threshold: *upper threshold* and *lower threshold*. Both of this values are compared against the current resource's *monitored value*. Whenever the monitored value from the resource surpasses the upper threshold, a scale out suggestion is made. In opposition, when the monitored value is under the lower threshold, a scale in suggestion is made. However, in order for a suggestion to become a scaling decision, other parameters must be taken into account. In our prototype, we utilize the number of pending messages in the RabbitMQ queue as a monitored value.

In the monitored resource element, the *status* is being constantly updated. A waiting time is introduced before each measurement, in order to avoid saturating scaling decisions. After each measurement, the resource's current status is updated. The status can be either overloaded, underloaded, or balanced, in regard to the threshold presented before. *Overloaded* status means that the upper threshold was surpassed, whereas the *underloaded* status means that the monitored value is below the lower threshold, lastly the *balanced* status means the monitored value is between both the lower and upper threshold. Moreover, to avoid taking rash decisions on temporary spikes in the monitored value, the scaling suggestion becomes an action, only after a specified number of contiguous measurements results in the same status. Similarly, after a scaling action has been taken, a *grace period* is introduced, in which the action taken can have a chance to show an impact on the monitored value. A pseudo-code of this interaction of elements can be seen in Algorithm 4.1.

Algorithm 4.1: Auto-Scaling

input : x, the waiting time before the next check
resource, the resource to check its status
gracePeriod, the time to wait before calling scale in/out again
succesiveChecks, the minimum number of checks with the same status
upper, the higher value of the threshold

```
1  n ← 0
2  status ← balanced
3  while true do
4  |  wait(x)                                   /* wait x milliseconds */
5  |  status ← fetchStatus(resource, upper, lower, n, status)
6  |  if lastStatus is overloaded and n > successiveChecks then
7  |  |  if is not in the gracePeriod then
8  |  |  |  scaleOut()
9  |  |  end
10 |  else if status is underloaded and n > successiveChecks then
11 |  |  if is not in the gracePeriod then
12 |  |  |  scaleIn()
13 |  |  end
14 |  end
15 end
16 Function fetchStatus(resouce, upper, lower, n, last)
17 |  value ← getValue(resource)
18 |  if lower > value then
19 |  |  status ← underloaded
20 |  else if upper < value then
21 |  |  status ← overloaded
22 |  else
23 |  |  status ← balanced
24 |  end
25 |
26 |  if last ≠ status then
27 |  |  n ← 0
28 |  else
29 |  |  n ← n + 1
30 |  end
31 |  last ← status
32 |  return last;
```

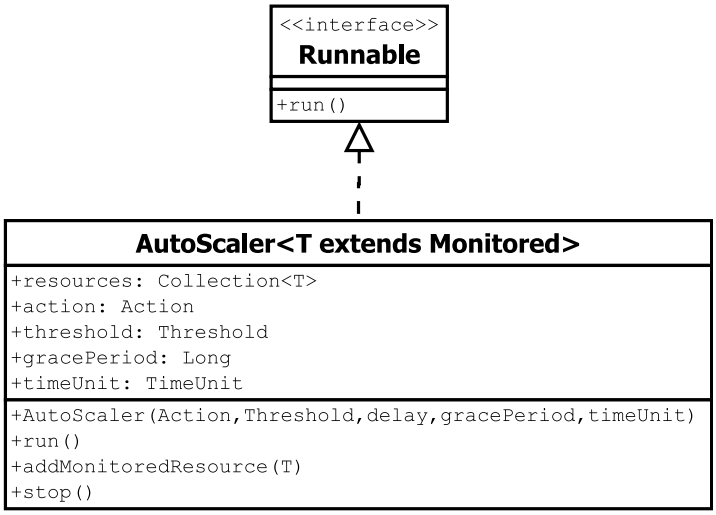


Figure 4.2: Auto Scaler class diagram.

4.2.1 Auto-Scaler implementation

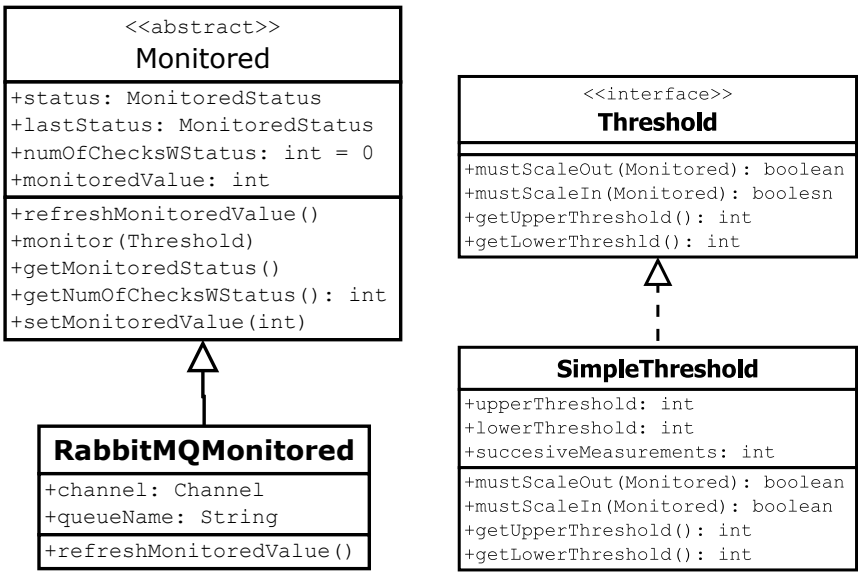
In this section, the implementation specifics of the Auto-Scaling algorithm will be presented. The class `AutoScaler` implements the `Runnable` interface as it will run concurrently while Storm process data (see Figure 4.2). As a waiting mechanism, the `Executors.newScheduledThreadPool(int)` is utilized, for ensuring the checks are made depending on the *delay* specified. In the implemented `run()` method, most of the Algorithm 4.1 is coded. However, in order to simplify the algorithm, some concepts were abstracted to make it more readable.

Monitored

The abstract class `Monitored` represents an element which will be monitored in terms of a given `Threshold` (see Figure 4.3a). In our case, the `RabbitMQMonitored` is in charge of refreshing the monitored value, so that it can be evaluated using the `monitor(Threshold)` method. The precondition for calling the latter is that the `AutoScaler` implementation calls the abstract method `refreshMonitoredValue()` before. Once this two steps are made in order, the `Threshold` implementation can check the status of the monitored component by calling `getMonitoredtatus()`. This method returns either overloaded, underloaded or balanced as values.

Threshold

The `Threshold` interface requires only four methods to be implemented, two of which are central to the scaling decision (see Figure 4.3b). The methods that decide whether to scale or not, are the `mustScaleOut(Monitored)` and `mustScaleIn(Monitored)` methods. Inside these methods, the status of the monitored resource is fetched by



(a) The Monitored abstract class and extending class RabbitMQMonitored.

(b) The Threshold interface and SimpleThreshold implementation.

Figure 4.3: Monitored class and Threshold interface class diagrams with implementations.

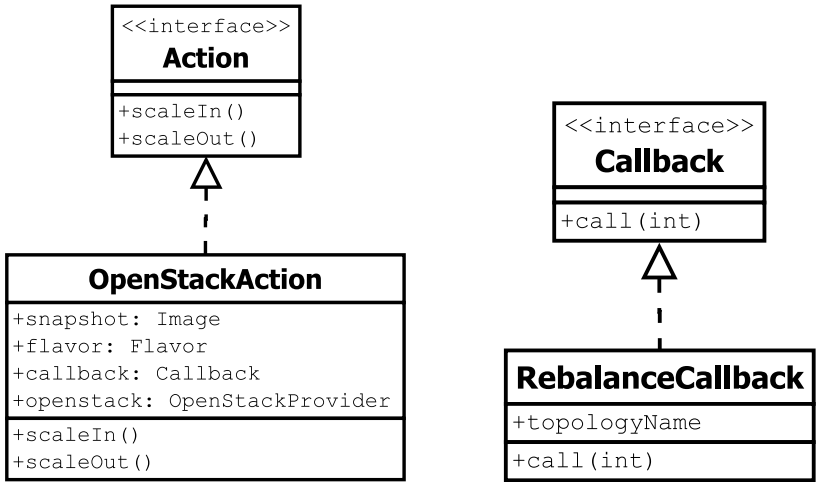
calling `getMonitoredStatus()` from the parameter. Subsequently the number of times this status has been retained on previous checks is retrieved by calling the method `getNumberOfChecksWithStatus()`. If the status is overloaded or underloaded and the value returned by the `getNumberOfChecksWithStatus()` is higher than configured, the called method will return `true`, otherwise if the value is lower, the method returns `false`.

Action

The `Action` interface, deals with the following steps after a decision from the `Threshold` has been made. This interface has two methods, `scaleIn()` and `scaleOut()` (see Figure 4.4a). A class implementing both methods should handle all of the necessary steps to achieve the action of scaling in or out. Specifically, the implementing class `OpenStackAction` deals with all of the RESTful API calls needed to create a new computing node. After the action has been fulfilled i.e., a new computing node is up and running, a callback method is then invoked.

Callback

The interface `Callback` provides a callback method with an integer as a parameter (See Figure 4.4b). This method has as a goal to make specific changes after a node has been successfully added. In our specific case, after adding a new node, the `RebalanceCallback` class, makes *rebalance* request to Apache Storm, with the new number of running nodes,



(a) The Action interface and implementing class OpenStackAction.

(b) The Callback interface and RebalanceCallback implementation.

Figure 4.4: Callback and Action interface class diagrams with implementations.

as well as new parallelization hints. The specifics of rebalancing in Storm can be found in Section 4.4.2.

4.3 Components

In this section, all of the utilized components in the prototype are going to be presented. For the prototype, OpenStack was selected as an infrastructure provider. Through the infrastructure provided by OpenStack, virtual machines will be made available to the Storm cluster on the fly. Each of these virtual machines will run some of the components later described. In Figure 4.5 a representation of the running virtual machines and their components can be found. A specific description of OpenStack will be given at Section 4.3.1.

In order to isolate the Apache Storm cluster, a computational node was added to run the components alien to Storm, shown in the figure as 'Frontend'. The Frontend node runs the RabbitMQ server, as well as the PostgreSQL database. The specific role of these technologies will be discussed in Section 4.3.3 and Section 4.3.2 respectively.

For Storm, at least two computational nodes are made available at all times, depicted as 'Coordinator' and 'Worker 1' in Figure 4.5. As its name suggests, the Coordinator node handles the coordination of the processing. To achieve that, Apache Storm needs two components Zookeeper[4] and Storm's Nimbus[5]. Finally, the worker nodes `(Worker 1, ... , Worker n)` only run Storm's Supervisor, which has the role of

[4]https://zookeeper.apache.org/

[5]http://storm.apache.org/releases/1.0.1/Setting-up-a-Storm-cluster.html

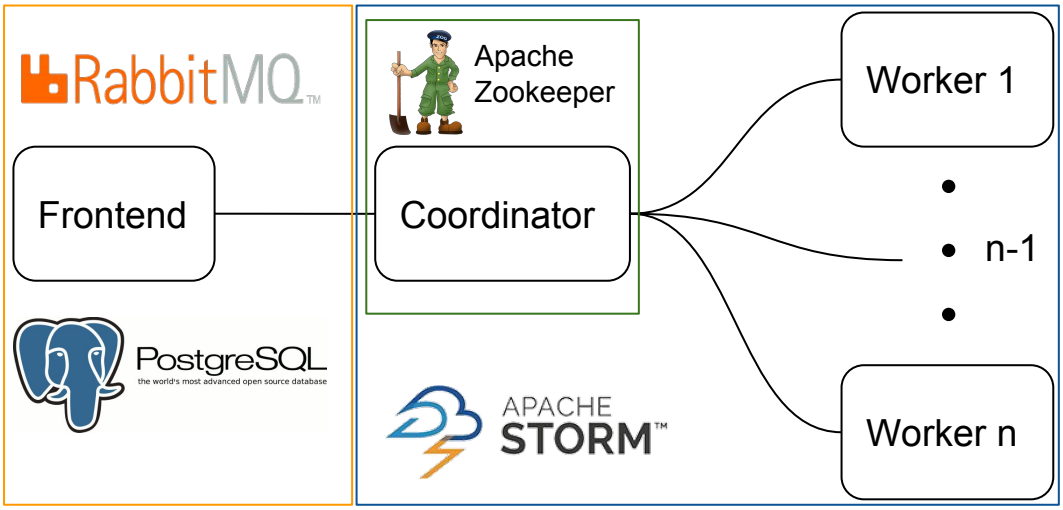


Figure 4.5: Computing Nodes

starting and stopping worker processes. Apache Storm will be thoroughly discussed in Section 4.4.

4.3.1 OpenStack

OpenStack is a cloud computing platform, utilized in this context to provide computational resources at runtime in the form of Virtual Machines. For each Virtual Machine at least a *name*, a *Flavor*, and an *Image* or *Snapshot* must be specified. The name specifies the machine's name and hostname. A Flavor specifies the amount RAM memory, VCPU (Virtual CPU's), and Disk space. Lastly, an Image typically specifies the Operating System to install on the first run, whereas a Snapshot is a copy of a running node at a specific time. The advantage Snapshots have against Images, is that a newly created node with a Snapshot will run as the copy. This has the potential of having ready prepared copies of running nodes' disks and operating system, allowing the creation of new nodes with them on demand, rather than having to set up each of them upon creation.

Virtual Machines

For all of our deployed computational nodes, the underlying operating system is *Ubuntu 14.04 LTS amd64*. However, each of the nodes have different Flavors. A comparison of the different Flavors utilized can be found in Table 4.2.

4.3.2 TDriveDatabase

The data presented in Section 4.1 has been persisted in a PostgreSQL server. This allows us to better organize the tuples by time order and request them easily. In order to

	Frontend	Coordinator	Worker
VCPU	3	2	1
RAM	5.76 GB	3.6 GB	1,8 GB
Disk	40 GB	40 GB	40 GB

Table 4.2: Comparison of Flavors of the Virtual Machines

transform the files organized by taxi id into persisted entities in the database, a tool was implemented named the *TDriveDatabase*.

This tool has two main functions: read line by line from the dataset files persisting them in the database, as well as reading entities from the database, sending them as they are being read to the RabbitMQ queue described in Section 4.3.3.

Alongside the two previous mentioned functions, a RESTful API was implemented in order to persist test run data. This data includes the starting timestamp of the topology, as well as a recurrent last updated timestamp. The data also includes the type of the deployed topology, together with a variable persisting whether the auto-scaler implementation was activated or not. Lastly, the report generated by the `GenerateReportBolt` discussed in Section 4.4.1 is also persisted.

4.3.3 RabbitMQ

RabbitMQ is a platform for exchanging messages between applications. In other words, the platform takes the role of a message receiver and carrier. When using RabbitMQ, there are three important components: the producer, the queue, and the consumer. The *producer* is the component that produces or writes a message, while the *consumer* consumes or reads them. In the time between creation and delivery, the message is stored in an unbound *queue*, thus guaranteeing order. This is the simplest model of a RabbitMQ queue, however, more complex configurations can be arranged[6].

In our prototype, there is no need to utilize a complex configuration of RabbitMQ; we utilize a simple work queue. The component described in Section 4.3.2, the *TDrive-Database*, is our producer filling the queue at a constant rate. The consumers, in this case, are Apache Storm's Spouts, which in turn feed tuples to Bolts for processing.

4.3.4 Starting the Auto-Scaler Component

In order to start the auto-scaling prototype, only two steps are required: calling the constructor method, adding a resource to monitor (See Figure 4.6). Once the Java object was constructed, the algorithm will start to constantly check the added monitored resources, as often as the parameter `threshlodCheckDelayInMilliseconds` specifies. Moreover, in the constructor the Action and Threshold objects are also specified. In

[6]http://www.rabbitmq.com/tutorials/tutorial-one-java.html

```
1 AutoScaler<T extends Monitored> as = new AutoScaler<>(Action action,
    Threshold threshold, Long thresholdCheckDelayInMilliseconds, Long
    gracePeriod, TimeUnit gracePeriodTimeUnit);
2 as.addMonitoredResource(Monitored);
```

Figure 4.6: Necessary Code to start the Auto-Scaler Component

this way, both the Action and the Threshold are abstracted from the implementation of the scaling algorithm. In addition to this, more than one monitored resource can be specified, this can be useful if more than just one parameter wants to be monitored.

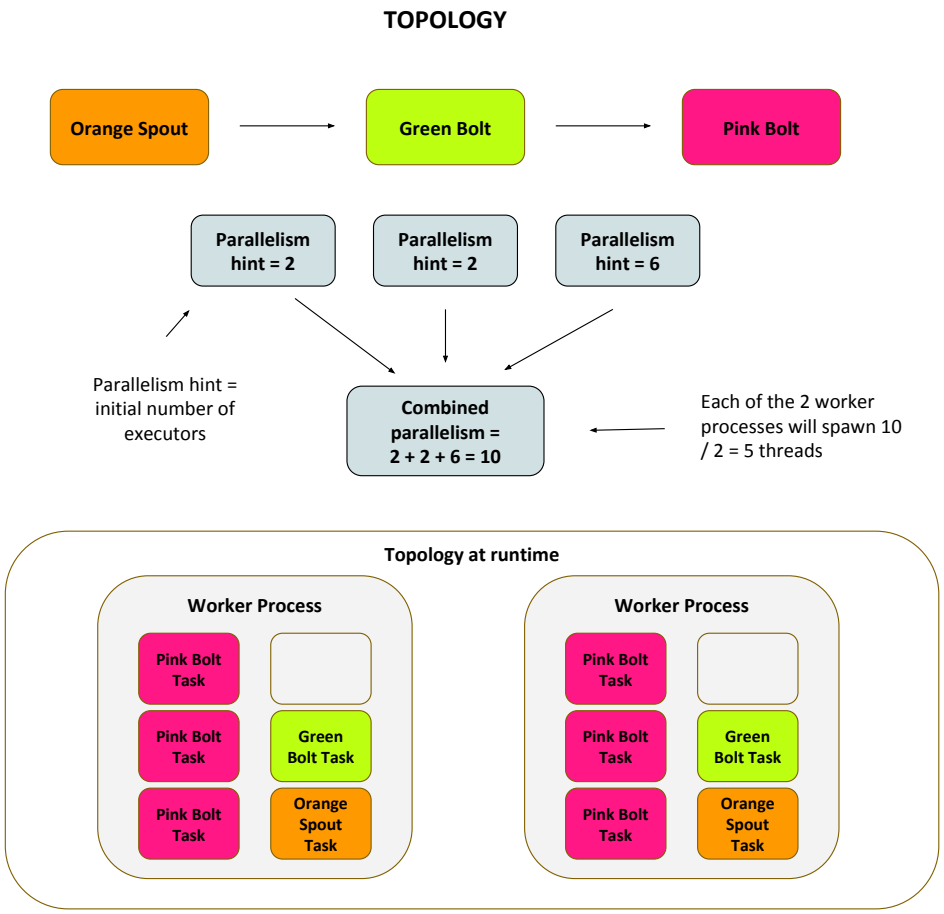
In our prototype, we included the code for starting the auto-scaling component in the same piece of code that submits a new topology in Storm. In this way, the algorithm would start alongside the topology, ensuring the monitoring of the progress from the beginning. However, with this approach, our prototype only supports the monitoring and scaling of one topology at any given time.

4.4 Scalability for Apache Storm

Apache Storm is an Open Source, real-time, scalable SPF. It is designed in a way, in which the developer can focus on the application logic. The application logic the developer produces is abstracted into a topology. A *Topology* is a directed graph, in which each of the nodes contains processing logic. This graph describes the way components are interconnected or arranged, specifically which nodes are subscribed to the emissions of other nodes. The graph includes *Spouts* and *Bolts* as nodes. A Spout is a data source that feeds tuples to Bolts. Whereas Bolts are consumers of tuples, in which processing takes place. Apart of consuming tuples, Bolts can also emit new tuples depending on the developer's requirements. This allows complex arrangements of Bolts and Spouts, allowing the creation of flexible topologies.

During the execution of a Storm Topology, multiple elements work together to process the incoming stream of tuples. In Storm, each of the nodes in the topology are converted into *tasks*. Each of the tasks in Storm are in charge of processing a part of the incoming tuples. Tasks are run by *executors*, tasks of the same Spout or Bolt can be grouped in the same executor. Storm executors are spawned threads by a *worker process* encapsulated in Java Virtual Machine. Multiple worker processes can be concurrently run inside any Supervisor node in a Storm cluster.

An example of a running topology can be seen in Figure 4.7. When submitting a topology in Storm, the number of parallel tasks can be defined using the `parallelism hint` parameter. The example shows one spout and two bolts with a total parallelism of 10. The Pink Bolt declared a parallelism of 6, whereas both the Green Bolt and Orange Spout declared a parallelism of 2. As two worker processes were also specified, 5 threads



Adopted from https://storm.apache.org/releases/1.0.1/Understanding-the-parallelism-of-a-Storm-topology.html

Figure 4.7: A Running Topology in Storm

will be spawned in each of the workers. If only one worker would have been available, all 10 threads would have been spawned in that single worker. The parallelism hint is specified when deploying the topology, however, it can be modified at runtime with the `rebalance` command.

The architecture of Storm includes three more components: *Supervisor*, *Nimbus*, and *ZooKeeper*. Each of this components has a specific task. The Supervisor is in charge of managing worker processes. There is exactly one Supervisor instance running at each processing host in the storm cluster. In order to run a storm cluster, at least one Nimbus instance is needed, as well as a ZooKeeper instance and the aforementioned Supervisor. The Nimbus instance is in charge of coordinating the execution of topologies. To achieve this, Nimbus includes a scheduler that defines the placement of the tasks in the available

worker nodes. To notify the worker nodes, Nimbus utilizes ZooKeeper's shared memory service for managing configurations and distributed coordination.

4.4.1 Deployed Topology

The following topology in Figure 4.8 has been deployed in the storm cluster, in order to evaluate the performance with our auto-scaling mechanism. We have utilized three different strategies regarding the distance calculation of the taxis. Namely the `TaxiDistanceBolt`, `TaxiNDistanceBolt`, and `TaxiWindowDistanceBolt`. At any given deployed topology, only one of this three bolts is active. This has as a goal, to evaluate Storm's capabilities with different aggregation strategies. Each of the three bolts exhibit a different behavior. The behavior of each of the bolts and the RabbitMQSpout are briefly discussed in the next paragraphs.

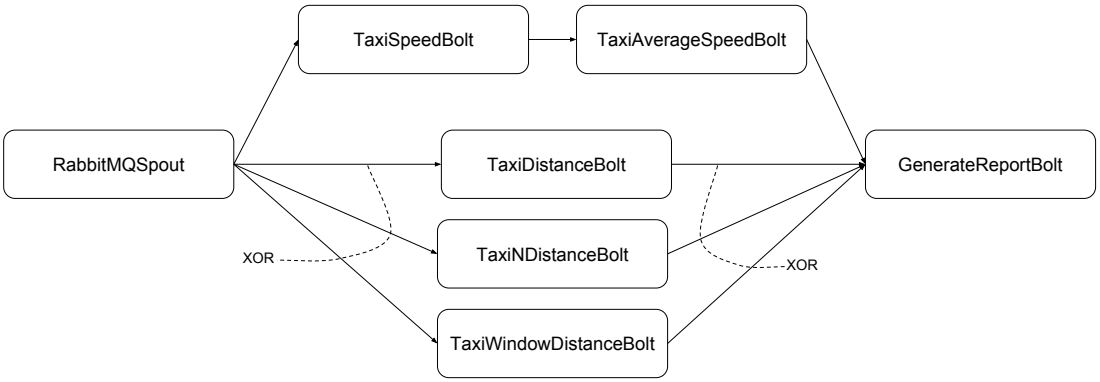


Figure 4.8: Deployed Storm Topology

RabbitMQSpout The `RabbitMQSpout` takes care of connecting to the RabbitMQ on the specified host and converting the messages consumed into tuples. Consequently feeding the subscribed bolts with these tuples.

TaxiSpeedBolt The `TaxiSpeedBolt` receives tuples from the `RabbitMQSpout`. In order to calculate the speed of the taxi, six values from two tuples of the same taxi are needed: the previous and current *date time*, *latitude* and *longitude*. For approximating to the real distance traveled by each of the taxis in the given timespan, due to Earth's spheric form, the haversine formula for calculating the distance between two points on a sphere was utilized:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{4.1}$$

Where r is the radius of the sphere, i.e., 6372.8KM for Earth. φ_1 and λ_1 are the latitude and longitude of the point of origin respectively, in radians. While φ_2 and λ_2 are the latitude and longitude of the last point respectively, also in radians.

Upon receiving the first tuple related to a taxi id, no calculation can be performed, as a minimum of two tuples is needed. The Bolt emits 0 and stores the values in a `Map`, so it can be retrieved upon the reception of a new tuple related to the same taxi. Should the needed data be present, the tuple then calculated speed using Galileo's speed formula:

$$v = \frac{d}{t} \tag{4.2}$$

The result of this calculation is then emitted, consequently replacing the last data stored in the `Map` with the calculated data.

TaxiAverageSpeedBolt The `TaxiAverageSpeedBolt` receives the tuples emitted by the `TaxiSpeedBolt`. Upon reception, the speed is added to a sum variable, and another variable with the number of items is increased. Both values are stored in a `Map` using taxi id as the key. Utilizing both the sum of received speeds and the number of items the average is calculated with the mean formula:

$$mean = \frac{1}{n}\sum_{i=1}^{n} a_i \tag{4.3}$$

Where n is the number of elements, and a_i is the sum of all the elements. After the calculation of the mean, the value is emitted alongside the taxi id.

TaxiDistanceBolt The `TaxiDistanceBolt` is subscribed to the `RabbitMQSpout`. Similarly to the `TaxiSpeedBolt`, the distance is calculated with the Formula 4.1. The previously calculated distance is then retrieved from a `Map` utilizing the tuple's taxi id, in order to add the newly calculated distance with the retrieved distance. This calculation results in the total distance traveled by this taxi. Consequently, the total distance is persisted again in the `Map`. For every received tuple, a new tuple is emitted with the total calculated distance.

TaxiNDistanceBolt The `TaxiNDistanceBolt` is subscribed to the tuples emitted by the `RabbitMQSpout`. In this bolt, the distance is calculated exactly as the `TaxiDistanceBolt`. However, instead of emitting a new tuple every time a new distance is calculated, it is emitted every 100 distance calculations. This reduces the number of emitted tuples hundredfold. However, it could be the case that we might not reach 100 tuples per taxi id. Therefore a timeout is included, so that after 60 seconds if 100 tuples were not reached, the total distance calculated for that taxi is then emitted.

TaxiWindowDistanceBolt As the previous two, the `TaxiWindowDistanceBolt` receives tuples from the `RabbitMQSpout`. This bolt utilizes the Windowing Support in Storm[7]. We have defined a tumbling window with a length of 5 seconds, in which all of the received tuples within that length are processed at once. After processing, the distances of the taxis that were modified are emitted.

[7] http://storm.apache.org/releases/1.0.0/Windowing.html

GenerateReportBolt The `GenerateReportBolt` is subscribed to the tuples emitted by the `Taxi*DistanceBolts`, as well as the `TaxiAverageSpeedBolt`'s emitted tuples as can be seen in Figure 4.1. This bolt receives all of the emitted distance and speed tuples, in order to create some metrics. The metrics generated are the top and bottom 5 taxis for both speed and distance. Every minute, this bolt sends all four lists to the `TDriveDatabase`'s REST API for persisting the results.

Stream Groupings

On the topology in Figure 4.9, when declaring a subscription between Bolts or Spouts, a stream grouping must be declared. A stream grouping defines how that stream should be partitioned among the bolt tasks. In Storm, there are eight built-in groupings, although a custom can also be implemented.

Shuffle grouping With this grouping, tuples are randomly distributed across the tasks, guaranteeing the same number of tuples among tasks.

Fields grouping With this grouping, tuples are grouped by the fields specified. This grouping guarantees that tuples with the same specified field, go to the same task.

Partial Key grouping This grouping works similarly as the fields grouping, although in contrast to fields grouping, this grouping balances the load between two downstream bolts.

All grouping With this grouping, all of the stream's tuples are replicated across all of the bolt's tasks.

Global grouping With this grouping, the entire stream goes to a single one of the bolt's task, the task with lowest id.

None grouping This grouping specifies no preference and works similarly as the shuffle grouping. However, Storm tries to move the execution of the task in the same thread as the bolt or spout they have subscribed to.

Direct grouping With this grouping, the produce of the tuple can decide to which task the tuple will be sent to.

Local or shuffle grouping With this grouping, if the target bolt has one or more tasks in the same worker process, the shuffling will occur only within those in the same process. Otherwise, it works just as the shuffle grouping.

In our specific case, in order to guarantee the calculations are made. We utilize the taxi id as our grouping field, as we need tuples of the same taxi id to reach the same task. Otherwise, there is no guarantee of correct calculations, as the tuples of that have to do with the same id, might not reach the executing task waiting for that data.

Deploying the Topology

In order to deploy the topology, the following command must be called on the coordinator node. This command submits a new topology, alongside the implementing `<topology-jar>` file.

```
storm jar <topology-jar>.jar <main-class> <topology-name> <opts>
```

Moreover, the `<main-class>` specifies the class to invoke upon submitting. A name for the topology is also required to be specified in `<topology-name>`. Lastly, in the `<opts>` optional configurations can be specified. In our implemented topology, the following options are available:

-rabbitHost This option allows to specify the IP address where RabbitMQ is running. Defaults to `localhost`.

-scaleGracePeriod This option allows to specify the scaling grace period in milliseconds, i.e., the time to wait before starting a new instance after the last was created. Defaults to 300000 milliseconds (5 minutes),

-measurementDelay This option allows to specify the delay, in milliseconds, between checking monitored resources. Defaults to 5000 milliseconds,

-upperThreshold This option allows to specify the upper threshold value. Whenever this threshold is surpassed, a scale out decision will be made. Defaults to 50000 units.

-lowerThreshold This option allows to specify the lower threshold value. Whenever the value is below this threshold, a scale in decision will be made. Defaults to 10000 units.

-successiveMeasurement This option allows to specify the number of successive measurements a resource must take before changing status. Defaults to 10 successive measurements.

-strategy This option allows to specify the strategy to be used. Can be `no-agg` for no aggregation of taxi distances, `sem-agg` for semantic aggregation, and `w-agg` for windowed aggregation. Defaults to `no-agg`. Each of this strategies selects the bolt to be used in the Topology shown in Figure 4.8. The `no-agg` parameter selects the `TaxiDistanceBolt`, whereas `sem-agg` selects `TaxiNDistanceBolt`, and lastly `w-agg` selects `TaxiWindowDistanceBolt`

-disableAutoScaler This option allows to disable the implemented auto-scaling mechanism. Defaults to enabled.

In Java, the topology is defined as seen in Figure 4.9, where the `SPOUT_PARALLELISM` variable is initialized with the integer value 3.

```
1  TopologyBuilder builder = new TopologyBuilder();
2
3  builder.setSpout(RABBITMQ_SPOUT, new RabbitMQSpout(rabbitHost, queueName),
       SPOUT_PARALLELISM);
4
5  builder.setBolt(SPEED_BOLT, new TaxiSpeedBolt(), BOLT_PARALLELISM)
6    .fieldsGrouping(RABBITMQ_SPOUT, new Fields(TAXI_ID));
7
8  builder.setBolt(AVERAGE_SPEED_BOLT, new TaxiAverageSpeedBolt(),
       BOLT_PARALLELISM)
9    .fieldsGrouping(SPEED_BOLT, new Fields(TAXI_ID));
10
11 switch(strategy){
12   case "no-agg":
13     builder.setBolt(TOTAL_DISTANCE_BOLT, new TaxiDistanceBolt(),
       BOLT_PARALLELISM)
14       .fieldsGrouping(RABBITMQ_SPOUT, new Fields(TAXI_ID));
15     break;
16   case "sem-agg":
17     builder.setBolt(TOTAL_DISTANCE_BOLT, new TaxiNDistanceBolt(),
       BOLT_PARALLELISM)
18       .fieldsGrouping(RABBITMQ_SPOUT, new Fields(TAXI_ID));
19     break;
20   case "w-agg":
21     builder.setBolt(TOTAL_DISTANCE_BOLT, new TaxiWindowDistanceBolt().
       withTumblingWindow(new BaseWindowedBolt.Duration(5, TimeUnit.SECONDS)),
       BOLT_PARALLELISM)
22       .fieldsGrouping(RABBITMQ_SPOUT, new Fields(TAXI_ID));
23     break;
24 }
25
26 builder.setBolt(REPORT_BOLT, new GenerateReportBolt("http://"+rabbitHost+"
       :9090/",runid), BOLT_PARALLELISM)
27   .shuffleGrouping(TOTAL_DISTANCE_BOLT)
28   .shuffleGrouping(AVERAGE_SPEED_BOLT);
```

Figure 4.9: Storm Topology Declaration in Java

4.4.2 Storm 'rebalance'

The core functionality utilized in our auto-scaling approach is Storm's `rebalance` command. This command allows to respecify the parallelism hint parameter discussed before, at runtime.

```
storm rebalance <toplogy-name> -n <num-of-workers> -e <task-name>=<parallelism hint>
```

After adding a new worker, a call to `rebalance` is made doubling the initial parallelism hint with the Java code in Figure 4.10. The only parameter of this function is the number of current running workers (`numOfWorkers`) so that the parallelism can be

```
Config c = new Config();
Map conf = Utils.readStormConfig();
c.putAll(conf);
NimbusClient cc = NimbusClient.getConfiguredClient(c);
Nimbus.Client client = cc.getClient();
RebalanceOptions options = new RebalanceOptions();
options.set_num_workers(numOfWorkers);
Map<String, Integer> executorOptions = new HashMap<>();
executorOptions.put(RABBITMQ_SPOUT, SPOUT_PARALLELISM * numOfWorkers);
int boltParallelism = BOLT_PARALLELISM * numOfWorkers;
executorOptions.put(AVERAGE_SPEED_BOLT, boltParallelism);
executorOptions.put(DISTANCE_BOLT, boltParallelism);
executorOptions.put(REPORT_BOLT, boltParallelism);
executorOptions.put(TOTAL_DISTANCE_BOLT, boltParallelism);
executorOptions.put(SPEED_BOLT, boltParallelism);
options.set_num_executors(executorOptions);
client.send_rebalance(topologyName, options);
```

Figure 4.10: Rebalance Call in Java

multiplied by that number. This is due to Strom's parallelism is uniformly distributed among the workers, as discussed in the example around Figure 4.7. Thus multiplying the starting parallelism coefficient, against the number of current running workers, would yield a doubling of the initially allocated computing resources. Unfortunately, Storm's `rebalance` calls for the destruction and re-creation of executors, losing thus all of the metrics gathered so far, and quite possibly losing aggregated values that have not yet been emitted.

4.5 Evaluation

In this section, we discuss how we evaluated the implemented prototype. As a point of comparison, two set of tests were conducted. One of the set was run with the auto-scaling algorithm disabled, producing thus a point of comparison for the next set of tests. In the second set of tests, the auto-scaling algorithm was enabled. Whenever both of the results are compared against each other, the impact of the algorithm can be perceived. In each of the set, three different tuple aggregation strategies were employed, with the goal of comparing the impact of the bolt's behavior. In the fist strategy, no aggregation of tuples was performed. This strategy will serve as a point of comparison for the next two. For the second strategy, a semantic aggregation was performed, reducing fivehundredfold the number of emitted tuples by the bolt. Lastly, in the third strategy, a window aggregation was performed utilizing Storm's built-in windowing support.

Throughout each test, every minute the number of processed tuples as well as the amount of leased workers for each of the aggregation strategies was measured and persisted. In

Figure 4.11 a comparison of the application's throughput with and without the auto-scaling algorithm utilizing no aggregation can be seen. Next in Figure 4.12 the same comparison can be found utilizing the semantic aggregation approach. Furthermore, a comparison for the window aggregation can be seen in Figure 4.13. Lastly, the number of leased worker nodes throughout the tests can be seen in Figure 4.14.

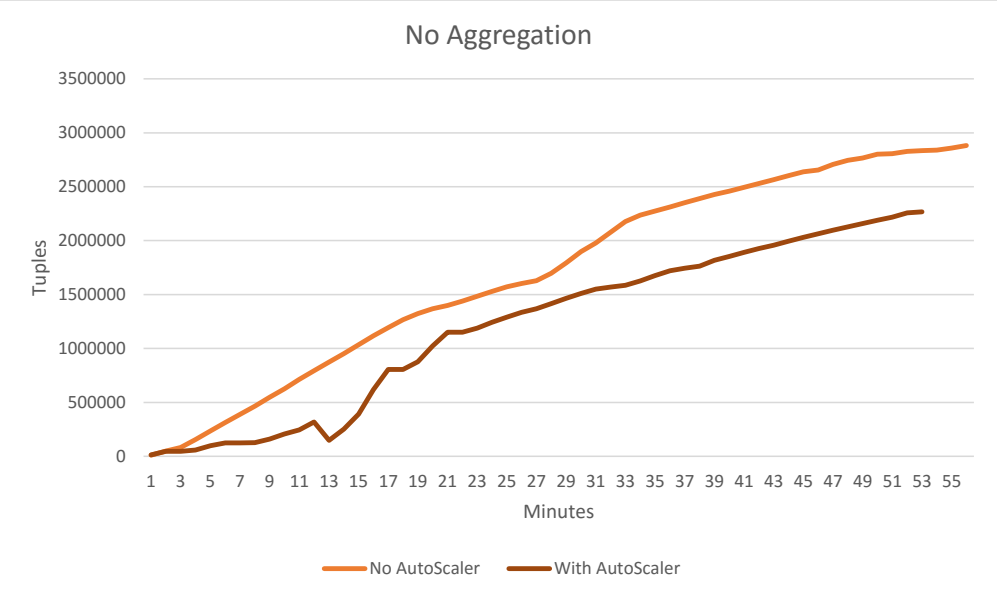


Figure 4.11: Amount of Tuples per Minute with No Aggregation

The dataset utilized discussed in Section 4.1, contains a total of 2,882,000 tuples. In some of our conducted tests, the application did not process all of the tuples. This can be due to Storm's rebalancing action requires Supervisors to restart their processing, possibly losing some of the tuples that were being processed. As a possible evidence of this, the loss of tuples can be consistently seen throughout the tests when the auto-scaling algorithm is enabled, consequently, the rebalancing action takes place at least eight times in some cases. Regardless, the chart in Figure 4.11 shows that our auto-scaling algorithm without utilizing an aggregation strategy did not increase the throughput of the application at any given time, rather there was a clear decrease in performance. This decrease in performance can also be seen utilizing the window aggregation strategy in Figure 4.13.

The performance decrease suggests that the rebalancing and scaling of Storm's topologies hinder the application's throughput. Our prototype reacts to the tuples waiting in the queue, however, due to fields grouping, if many of these tuples have the same taxi id, they would be grouped in the same executor, regardless of the total capacity. This grouping can result in having overloaded nodes while the rest are underloaded, reducing thus the throughput. In our dataset, we have 24 different taxis. These taxis would then be partitioned between the executors, at first we have a parallelism of three, making thus 8

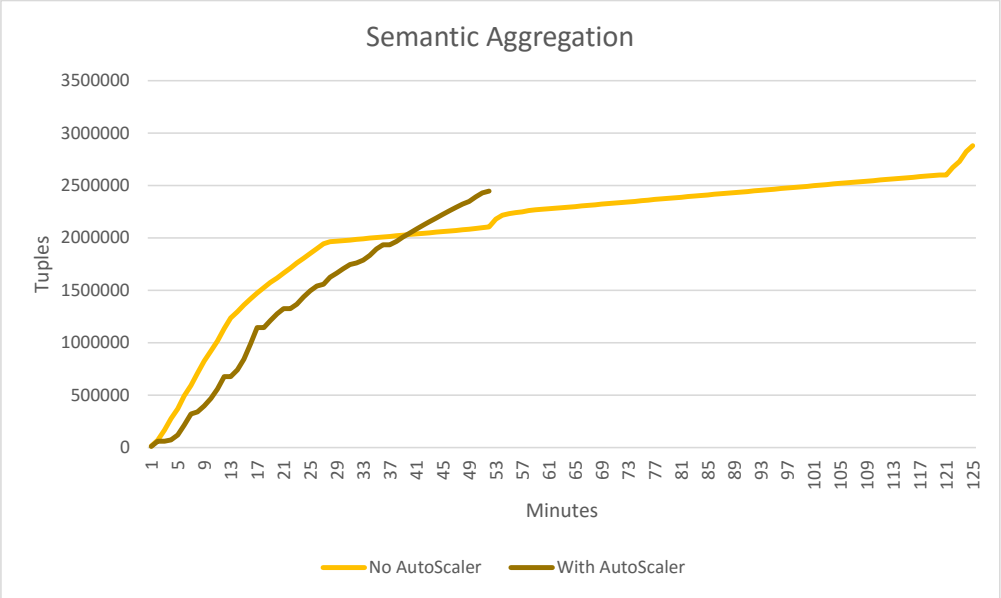


Figure 4.12: Amount of Tuples per Minute with Semantic Aggregation

taxis per executor. Whenever the topology scales out, this number would decrease in function to the parallelism number. For example, when our auto-scaling algorithm offers 3 workers nodes, the parallelism number would increase to 9, having thus around 3 taxis per executor. Should the number of tuples of a certain taxi id be significantly larger than the rest, scaling out would have no impact on their processing. Therefore, scaling topologies utilizing fields grouping with a relatively small amount of different taxis to process shows no impact on the applications throughput. Moreover, the stopping and resuming of the executors reduce the effectivity of the application due to the lost tuples and the induced overhead.

In the semantic aggregation, shown in Figure 4.12, the processing throughput is increased after minute 37, in contrast to the test without the auto-scaling algorithm. The semantic aggregation collects five hundred tuples for producing one and if every 30 seconds that number was not reached, the tuples received would be processed anyways. However, it could have been the case that the 500 tuples were rarely reached, therefore imposing a 30 second delay on the processing of each taxi id. Delaying the complete processing of the tuples. However, this appears to not be the case with the auto-scaling algorithm. This suggests that Storm's rebalancing command was beneficial to this strategy.

Comparing the first and the second aggregation strategies, the second showed a clear performance degradation due to the strategy employed. This suggests that the strategy was not correctly designed, and needs to be revised. Moreover, the window aggregation had also a slight performance decrease, suggesting that our windowing strategy was not properly designed.

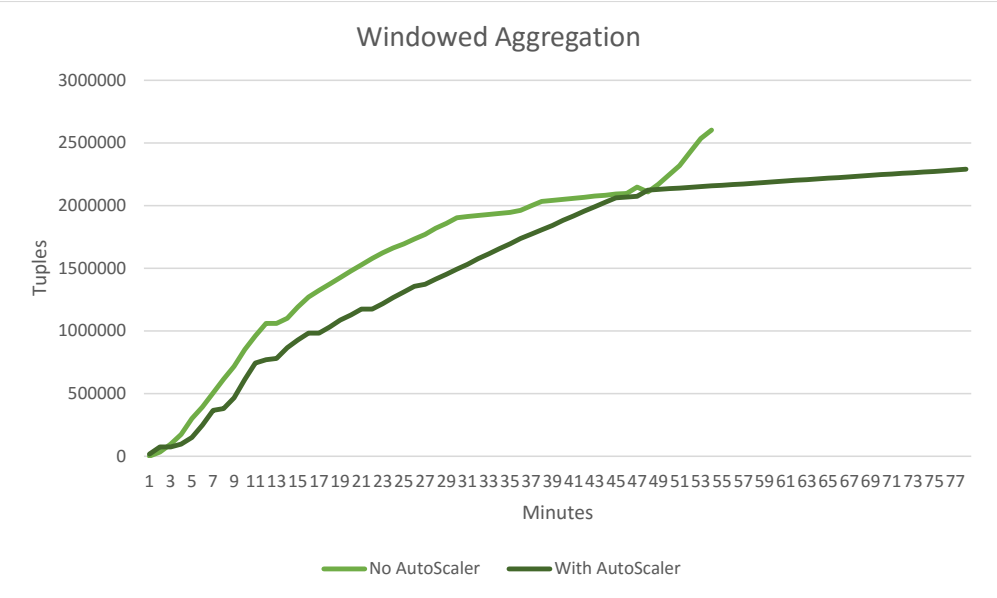


Figure 4.13: Amount of Tuples per Minute with Window Aggregation

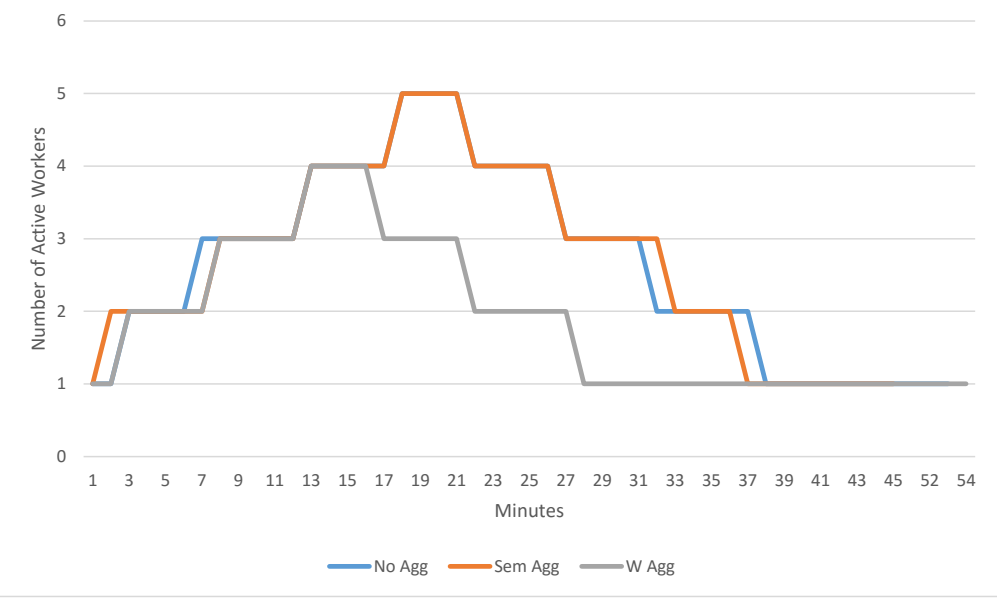


Figure 4.14: Amount of Leased Workers per Minute

CHAPTER 5

Discussion

We presented a working prototype with the goal of adapting to volatile data streams utilizing a global threshold algorithm. Our results suggest that more work needs to be done for effectively scaling arbitrary Storm topologies Moreover, the prototype has been tested in a controlled environment, somewhat far from real conditions. In order to have deterministic results, a mostly constant data rate was utilized, where the number of processed tuples and the number of leased workers was evaluated over time. Moreover, the scaling parameters were not explored for obtaining optimal decisions. Finding optimal parameters can positively influence the scaling decisions of the algorithm, possibly yielding better results. Regardless, the goal of this thesis was not to find an optimal scaling strategy, rather identify how and in which way could Storm and other SPFs deal with varying data streams. Therefore we revisit the research questions presented in Section 3.1.

1. **Which approaches are possible to deal with a varying amount of streaming data?** Apart from identifying multiple approaches, like load shedding and load balancing, we concentrated in resource elasticity at run-time utilizing an auto-scaling mechanism to add or remove computing hosts. Whenever new resources were added, the load balancing algorithm had the task to automatically relocate existing tasks as well as creating new ones. As our results in the previous section indicate, more work needs to be done for having an impact through an auto-scaling mechanism of this type on the application's throughput, as well as possibly providing a resource-effective computation.

2. **How are state of the art SPFs capable of dealing with varying amounts of streaming data?** Apache Storm has a built-in load balancing strategies, that depending on the user's demands, it balances the load differently among distributed tasks. Although Storm's load balancing strategies seem effective, they will eventually be constrained by the underlying systems processing capacity, due to their lack of horizontal resource elasticity. In our evaluation, we have seen how

Storm is able to adapt to varying amounts of streaming data with the help of our auto-scaling prototype. Our approach of adding and removing computing hosts, together with Storm's `rebalance` command, did not show a positive impact on the application's throughput. This could be due to the way tasks are balanced with the fields grouping.

In the scope of investigating this two questions, two important research topics were also identified. The scaling of stateful stream processing and the way auto-scaling decisions are made. Scaling stateful stream processing is no easy task, as states must be migrated as resources become available. This action of migration can take a toll on the network and in the global throughput of the application. We have not identified a SPF that provides extensive support on this topic.

The second research topic is when to make scaling decisions and in respect to which parameters. Scaling too often, or scaling too little are possible negative scenarios, which depending on the SPF, or if states migration will take place, the global throughput could be substantially hindered if too many close together scaling decisions are made. In our prototype, a global threshold algorithm was utilized, taking into account only the number of pending messages in the RabbitMQ queue. Our approach does not take into consideration, whether some of the computing hosts are overloaded. Even if the message queue is not full, having an overloaded host might lead to a loss of throughput, due to the fact that storm only allows a maximum number of concurrent tuples at any given time and does not rebalance automatically at run-time.

Lastly, selecting a proper aggregation strategy can have a significant impact on the system's throughput. With an aggregation strategy, the number of tuples can be decreased, possibly reducing the capacity overload. However, utilizing an aggregation improperly can also have a negative impact. In our preliminary tests, we utilized a sliding window of 50 tuples. This sliding window created a significant performance degradation. We suspect that utilizing field grouping together with a sliding window, was creating a bottleneck, as it could have been the case that for that specific task, no tuples of the assigned groups were received. Resulting in the task to wait until tuples were received.

CHAPTER 6

Conclusion

The need to process large amounts of streaming data in real-time is becoming larger. With the growing of the Internet of Things, the amount of streaming data available has increased, and will continue to increase as more and more devices become interconnected. Moreover, the need of real-time information has also become apparent with the attention SPFs have received. Providing a load balancing scheme, regardless of its effectivity, will eventually be constrained by the physical resources allocated in the computing hosts. Therefore SPFs must be able to provide a way to automatically scale horizontally the applications developed with them. Our approach did not provide conclusive results of the impact in the throughput when automatically scaling applications. However more work needs to be done, for evaluating auto-scaling approaches for stream processing, due to the potential of removing the constraint placed by finite physical resources of static computing clusters.

Auto-scaling stream processing applications seem promising, however, our tests were conducted in stateless processing, meaning that there was no state was preserved. In stateful processing, states must be migrated and can take a toll on the application's throughput. Whenever an application decides to scale, the toll on migration must be taken into account, so that the potential gain by adding new resources, will not be affected by the migration of states. On the contrary, stateless processing can scale without regard to this trade-off. However, apart from taking into account the consequences of a scaling decisions, the parameters to monitor must also be carefully selected. In our prototype, a global threshold algorithm was utilized, taking into account only the number of pending messages in the RabbitMQ. A local threshold algorithm would take into account the resources utilization of each of the computing hosts. Monitoring local resource allows a more comprehensive covering of the current capacity, whereas the monitoring a global one does not react on individual overloaded computing nodes. Other metrics must also be in place to take into account the requirements set by service line agreements. Looking back previously mentioned challenges for auto-scaling techniques by Heinze et. al. [HPJF14],

we conclude that our prototype proposed solutions for the first two challenges. Avoiding to have fixed workload models was achieved utilizing Storm's `rebalance` command, as well as OpenStack's virtual machines. Although the SPF's capacity was increased, due to the grouping utilized, there was no positive impact on the application's throughput.

Aggregation strategies can help reduce the number of tuples to be processed. We evaluated two aggregation strategies were performed: the semantic aggregation, and the window aggregation. However, if an aggregation strategy is designed without care, it can have a negative impact, as the aggregation can act as a bottleneck. In our preliminary tests, a badly selected windowing scheme significantly reduced the application's throughput. Moreover, our semantic aggregation's approach also had a significant negative impact on the throughput, due to it's configuration.

We successfully integrated Storm with OpenStack for dealing with Hummer et. al.'s [HSD13] first challenge of Elastic SPFs: have an optimized integration with cloud environments. In regard to the other three challenges, Storm is only a Framework and does not provide stream processing as a service, neither provides a monitoring infrastructure, nor context-and-locality aware data delivery. All in all, we can conclude that Apache Storm has the possibility to integrate an auto-scaling algorithm and substantially increase the throughput of the developed applications.

Bibliography

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Conference on Innovative Data Systems Research*, volume 5, pages 277–289, 2005.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Technical Report No. UCB/EECS-2009-28*, 2009.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[ALHL14] Hanieh Alipour, Yan Liu, and Abdelwahab Hamou-Lhadj. Analyzing auto-scaling issues in cloud environments. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 75–89. IBM Corp., 2014.

[BBC+04] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, et al. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[CGLPN15] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[CGLPN16] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80. ACM, 2016.

[CNL16] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. Elastic stateful stream processing in storm. *Proceedings of the 2016 International Conference on High Performance Computing & Simulation*, 2016.

[DGST11] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *IEEE Internet Computing*, (5):66–71, 2011.

[GJPPM+12] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[HBS+16] Kai Hwang, Xiaoying Bai, Yue Shi, Muyang Li, Wen-Guang Chen, and Yongwei Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):130–143, 2016.

[HPJF14] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *IEEE 30th International Conference on Data Engineering Workshops*, pages 296–302. IEEE, 2014.

[HSD13] Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013.

[HSDL15] Christoph Hochreiner, Stefan Schulte, Schahram Dustdar, and Freddy Lécué. Elastic stream processing for distributed environments. *IEEE Internet Computing*, (6):54–59, 2015.

[HVSD16] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. Elastic stream processing for the internet of things. In *9th International Conference on Cloud Computing*, pages NN–NN. IEEE, 2016.

[HVWD16] Christoph Hochreiner, Michael Vögler, Philipp Waibel, and Schahram Dustdar. VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things. In *20th Int. Enterprise Distributed Object Computing Conference*, pages NN–NN. IEEE, 2016.

[IS11] Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *IEEE International Conference on Cloud Computing*, pages 195–202. IEEE, 2011.

[KKP11] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter Pietzuch. Balancing load in stream processing with the cloud. In *2011 IEEE 27th International Conference on Data Engineering Workshops*, pages 16–21. IEEE, 2011.

[LTDH+14] Freddy Lécué, Simone Tallevi-Diotallevi, Jer Hayes, Robert Tucker, Veli Bicer, Marco Luca Sbodio, and Pierpaolo Tommasi. Star-city: semantic traffic analytics and reasoning for city. In *Proceedings of the 19th international conference on Intelligent User Interfaces*, pages 179–188. ACM, 2014.

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. *Special Publication (NIST SP) - 800-145*, 2011.

[MSDPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[PS06] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *International Conference on Extending Database Technology*, pages 445–464. Springer, 2006.

[SAG+09] Scott Schneider, Henrique Andrade, Buğra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[SeZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[TAe+06] Nesime Tatbul, Yanif Ahmad, Uğur Çetintemel, Jeong-Hyon Hwang, Ying Xing, and Stan Zdonik. Load management and high availability in the borealis distributed stream processing engine. In *International conference on GeoSensor Networks*, pages 66–85. Springer, 2006.

[TeZ+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases*, number Volume 29, pages 309–320. VLDB Endowment, 2003.

[TeZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[TZ06] Nesime Tatbul and Stan Zdonik. Dealing with overload in distributed stream processing systems. In *22nd International Conference on Data Engineering Workshops*, pages 24–24. IEEE, 2006.

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[WT15] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic stateful stream computation in the cloud. In *IEEE 31st International Conference on Data Engineerin*, pages 723–734. IEEE, 2015.

[Zhe11] Yu Zheng. T-drive trajectory data sample. [Online] Available: http://research.microsoft.com/apps/pubs/default.aspx?id=152883, August 2011. Accessed on 20.07.2016.