Distributed Systems for Processing Large Scale Data Streams

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatikerin

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Magdalena Soyka
geboren am: 14.12.1979
in: Berlin

Gutachter(innen): Prof. Freytag, PhD
Prof. Dr. Leser

eingereicht am: ......
# Contents

1 Introduction ................................. 1
   1.1 Motivation ................................ 1
   1.2 Objective ................................ 2
   1.3 Outline .................................. 2

2 Preliminaries ............................... 3
   2.1 Parallel and Distributed Computing .......... 3
   2.2 Batch Processing and Stream Processing .... 4
   2.3 Scalability & Fault Tolerance ............... 5
   2.4 Requirements for Real-Time Processing ...... 5
   2.5 Running Example: Linear Road Benchmark .. 6

3 Background Concepts and Related Systems .... 9
   3.1 Parallel Batch Processing ................. 9
      3.1.1 MapReduce ........................... 9
      3.1.2 Apache Hadoop ....................... 12
      3.1.3 MapReduce-like Systems .......... 14
   3.2 Data Stream Processing .................... 16
      3.2.1 Data Stream Model .................. 16
      3.2.2 Continuous Queries ................ 17
      3.2.3 STREAM ............................ 18
      3.2.4 Aurora ............................. 19
      3.2.5 Other Systems ...................... 21
   3.3 Parallel and Distributed Stream Processing .. 22
      3.3.1 Challenges ......................... 22
      3.3.2 Distributed Stream Processing Systems .. 23
   3.4 Summary .................................. 24

4 Case Studies .............................. 25
   4.1 Method and Set of Criteria ................. 25
   4.2 Storm .................................... 27
      4.2.1 Main Concepts ....................... 27
      4.2.2 Architecture ......................... 28
      4.2.3 Programming Model .................... 29
      4.2.4 Parallelism & Scalability ............ 30
      4.2.5 Fault Tolerance & Reliability ........ 31
      4.2.6 Extensions .......................... 33
   4.3 Apache S4 ................................ 34
## CONTENTS

4.3.1 Main Concepts .................................................. 34  
4.3.2 Architecture ................................................... 35  
4.3.3 Programming Model ........................................... 36  
4.3.4 Parallelism & Scalability ..................................... 37  
4.3.5 Fault Tolerance & Reliability ................................ 38  
4.4 Summary ............................................................ 40  

5 Experimental Evaluation ........................................... 42  
5.1 Methodology ........................................................ 42  
5.2 Limitations .......................................................... 44  
5.3 LRB Implementation ............................................... 44  
5.4 Results ............................................................... 46  

6 Conclusion ............................................................ 48
# List of Figures

2.1 Querying static data & Querying data in motion. .......................... 4
3.1 A MapReduce example program that computes sum and average of tolls. .... 10
3.2 An exemplary MapReduce Data Flow. ........................................ 11
3.3 Query plans in STREAM. .................................................... 20
3.4 Aurora Query Model. ......................................................... 21
3.5 Overview of presented Systems. ............................................. 24
4.1 Conceptual view of a Storm topology. ...................................... 28
4.2 Storm Components. .......................................................... 29
4.3 Rebalancing a Storm topology. .............................................. 31
4.4 Tuple tracking in Storm. ..................................................... 32
4.5 S4 app example. ............................................................... 35
4.6 Components of a Processing Node in S4. .................................. 36
4.7 S4 Partitioning Approaches. ................................................ 38
4.8 Checkpointing in S4 .......................................................... 39
5.1 LRB - Load distribution for $L = 1$. ..................................... 43
5.2 StormLRB Linear Road Benchmark Implementation. ..................... 45
5.3 StormLRB application with two partitioning approaches. ................... 45
5.4 S4 Linear Road Benchmark Implementation. ................................ 46
5.5 Latencies of StormLRB for $L=2$. ......................................... 47
List of Tables

4.1 Comparison of Storm and S4. .................................................. 41
5.1 Examplary values for latencies when running the benchmark in S4 and Storm. . 47
Chapter 1

Introduction

1.1 Motivation

The amount of newly generated data is increasingly growing in various areas and forms: transactions in trading, log data from servers, e-mails, tweets, blogs, photos and videos from web users, and sensor and measurement data from scientific experiments. Big Data has become the buzz word of 21st century large scale computing, especially in the commercial world. The term does not have a fixed definition, but is often broken down by three characteristics: Volume, Velocity and Variety. The so called 3 Vs reflect quite convenient the interdependent dimensions of Big Data. For instance, one can manage a huge amount of very simple structured data or one can have a relatively small amount of infinitely incoming disparate, complex, possibly unstructured data which additionally needs to be processed within strictly guaranteed periods of time, that is in real-time. Processing of these vast amounts of data can only be achieved by distributing the workload across many machines and implementing suitable programming paradigms. The introduction of the programming model of MapReduce [DG08] belongs to those kind of modern approaches. It enables an easy, efficient, highly scalable and fault tolerant way of parallelizing and distributing processing of large scale data.

However, dealing effectively with Big Data often requires to process data with its volume and variety while it is still in motion, not just after it is at rest. The demand for systems which process data in (near) real-time is constantly growing. Such systems share the common goal to continuously process incoming data. Systems, however, based on MapReduce such as Hadoop [Hada] are batch systems and not designed for processing data streams.

To address the velocity, technologies like active databases, publish-subscribe systems or complex event processing systems (CEP) emerged. In the academic world, general-purpose data stream management systems (DSMS) such as STREAM [Ara+04b] and Aurora [Aba+03] and TelegraphCQ [S.+03] were developed, but focused on velocity and could not efficiently process large amounts of data streams. Only in the last few years, with the development of systems like Borealis [Aba+05] in academia and IBM® InfoSpheres [Inf] in the commercial world, a new class of distributed stream processing systems (DSPS) was introduced.

In the open source world, systems for developing distributed real-time stream processing applications like Apache Storm [Sto] and Apache S4 [NRNK10] have been developed, facilitating scalable parallel processing of very large data streams. These two system will be the objects of this investigations. Both have already been in extensive use in Internet applications!
CHAPTER 1. INTRODUCTION

1.2 Objective

This thesis aims at investigating recent distributed stream processing systems. The requirements for systems that are capable of continuously processing these amounts of data differ from those of traditional approaches. Which requirements these are and what approaches the systems take to ensure their fulfillment will be established. The question, how those systems fit into the landscape of large scale data processing, is also discussed. Even though the systems share a common goal to process large scale data streams with low latency, they differ considerably in particular aspects such as architecture, data model and programming model. These distinctions and common features will be determined, while highlighting strengths and weaknesses. The main focus will be on the following aspects:

- Scalability: How is scalability ensured?
- Parallelization: How is workload parallelized and distributed?
- Fault Tolerance: How is reliability and robustness ensured?

The systems in question will be evaluated more closely in order to see what impact the design choices have on their performance. By implementing the Linear Road Benchmark [Ara+04b] a comparative performance measurement will be presented. Scalability shall be compared by running the benchmark on different distributed configurations. Before presenting the systems and the results of the benchmark an overview of the large scale data processing landscape is given.

1.3 Outline

At the beginning, basic terms and definitions that constitute the area of large scale processing will be introduced. One section will deal with the application described by the Linear Road Benchmark. This will be used as an example throughout this work. The chapter on preliminaries will be followed by an account of basic concepts, which essentially reflect a literature review focusing on batch and stream processing. First, MapReduce will be introduced. Subsequently, an outline of the stream paradigm will be provided. Both parts will involve descriptions of the most important or most commonly used systems for their respective fields. After that, an analysis of the main requirements for scalability, fault tolerance and parallelization will be given and exemplified by the two systems Apache S4 and Apache Storm. Finally, the findings resulting from the evaluation will be summed up and assessed, concluding this survey.
Chapter 2

Preliminaries

Aiming at providing the underlying key concepts of the investigation, this chapter will introduce a short overview of the definitions and relations of the basic terms Parallel and Distributed Computing as well as Batch and Stream Processing. Subsequently, two main requirements – Scalability and Fault Tolerance – will be explained in their distinctive shaping of existing distributed stream processing systems since these criteria of efficiency are of main concern to the following case studies. They are connected to a list of requirements for these systems that will then be presented. Ultimately, I will present the application described by the Linear Road Benchmark, which was used for the performance evaluation concluding the study, as it serves as a running example to illustrate some concepts in this work.

2.1 Parallel and Distributed Computing

Parallel computing has been an active research area for decades [KMZS08]. Given that the potential of improving performance by increasing CPU clock speed is limited much of research has been done concerning the development of multi-core architectures involving an increased performance by enabling parallel computing. Today’s multi-processor machines may be equipped with a few terabyte of disk storage. However their I/O performance is limited. Thus even a powerful computer is undersized for problems in the petabyte scale. The only way to achieve higher performance for large scale processing is to exploit parallelism with multiple machines. Whereas historically, parallel computing used to be restricted to shared-memory or shared-disk architectures, nowadays clusters comprised of commodity hardware are the dominant architecture in data centers in the Big Data domain. Such distributed systems which thoroughly depend on message passing for communication, can achieve a much higher parallelism than shared architectures because they are not limited by sharing channels, I/O operations or memory size [Sto86].

For parallel programming there are two types of parallelism: data and task parallelism. Data parallelism relates to scenarios, in which one task is executed simultaneously on individual parts of the data. Problems that are related but can be processed independent of each other are called embarrassingly parallel. MapReduce, which will be introduced in the following chapter, is an example of a programming model which makes use of the data parallelism paradigm. Task parallelism is the concurrent execution of different tasks or functions on the same or different data. Pipeline parallelism is a specialization of task parallelism and refers to the concurrent execution of two tasks that constitute a pipeline, e.g a first task as a producer of the input of the second task. When querying data further aspects can be considered for parallelization: Inter-query parallelism
refers to the parallel execution of several different queries and *intra-query* parallelism refers to parallelization of operators within a query. The latter involves two complementary approaches for realizing parallelization: *inter-operator parallelism*, by executing several different operators in a query expression in parallel as opposed to *intra-operator parallelism* where the execution of same operator is parallelized in the query [OV11]. The development of concurrent applications has been a challenge for years. Indeed there is no easy solution. Meanwhile different ways to address this problem have been developed. The *Actor Model* represents one way of implementing concurrent algorithms. An actor is an active object whose code is executed by exactly one process. Each actor is supervised and communicates with other actors only via asynchronous messaging, which enables robustness. This encapsulation makes the model inherently distributed while being highly scalable [Agh86].

### 2.2 Batch Processing and Stream Processing

The purpose of any Big Data solution is to convert data into relevant information. Recent applications in large scale data processing utilizing MapReduce focus on *batch processing*. Batch processing is a term to describe a form of data processing with a minimum of human interaction. The term originates from the early days of computing when programs and/or data sets were entered into the computer and processed as a coherent “batch” often in several consecutive program sequences. Batch processing, however, has significant downsides. Every job must be run through to completion. Changes in the data require reprocessing of the batch job itself. Depending on the amount of data and the computational power, this may lead to high latencies in processing. However, more and more applications require processing in *real-time*. Real-time processing both refers to processing a query with a *low latency* and to continuously processing incoming data as soon as it arrives. The latter is referred to as *stream processing*. MapReduce is not feasible for either case. Especially if the input rate at which data is arriving has a high velocity. The main difference between both approaches is: Whereas in batch processing a query is executed once over *static data* (or *data at rest*), in stream processing a *continuous query* or *standing query* is issued once and repeatedly evaluated for an indeterminate period of time over a data stream [BBDMW02]. The former will produce only one result while the latter will continuously output updated results. Figure 2.1 illustrates this paradigm shift.

![Figure 2.1: Querying static data & Querying data streams ([Rea10]).](image)

Depending on the field, real-time can mean different things. Due to the amount of data to be analyzed and the scope of the application, an appropriate latency for realizing a real-time or near real-time system can be microseconds with e.g. high-frequency trading, milliseconds when processing the output of the Large Hadron Collider or seconds with everyday applications such as search engines, recommendation systems in online shops and fraud detection. That said, even time spans of minutes, hours or days can still be considered as low latency e.g. in science with for example genome sequencing, climate simulations or satellite image processing. In these cases the data is
so huge that even supercomputers still need that much time. This is why there is a distinction between *hard* and *soft* real-time conditions. Hard real-time conditions refer to a situation in which a delayed execution may result in damage, e.g. in a crashing car, where processing data of multiple sensors trigger the inflation of an airbag as opposed to e.g. analyzing user behavior whilst online shopping to compute recommendations. Given that latency is subject to too many factors, such as bandwidth, routing hardware, or performance of the connected computers, for example, only soft conditions are ensured in practice.

### 2.3 Scalability & Fault Tolerance

It is the goal of a distributed architecture to achieve a scalable infrastructure. *Scalability* refers to the ability of a system to cope with quantitatively increasing demands. A system can be scaled vertically and horizontally. [*Scaling vertically* (scale up)] means increasing the resources of the elements of a system like buying a new supercomputer, adding more memory or updating CPU or even replacing the machine with a newer model. Whereas [*scaling horizontally* (scale out)] means adding more processing elements to a system such as adding more machines to a cluster [Pav+09]. From the developer’s point of view it is easier to scale vertically, which is more expensive, however, and the benefits are short-termed as well as limited by the available technology. Horizontal scaling is theoretically unlimited, but applications for scalable distributed systems are more difficult to design and monitor, than centralized systems, especially regarding clusters comprised of commodity machines.

Such an environment comes with several challenges for developers. One of the main challenges is the occurrence of faults. There are many sources of failures such as hardware and software failures, unavailability due to overload of a component or blackouts. In a distributed setting, especially when working with a large number of commodity machines, errors are predestined to happen [Pav+09]. For a distributed system to be reliable and robust, it is therefore of importance to be fault tolerant. *Fault tolerance* refers to the ability to detect and (automatically) recover from component failures.

### 2.4 Requirements for Real-Time Processing

Historically a lot of the systems for processing data streams were custom systems specifically developed for an application. As of this millennium, however, more and more general purpose solutions began to emerge. In this context the authors of [ScZ05] present eight rules characterizing the requirements for real-time stream processing systems.

**Rule #1: In-stream Processing.** The System has to be able to process arriving data items *in-stream* (on the fly) without constantly using a storage operation to avoid unnecessary latencies.

**Rule #2: Supporting a high-level query language.** The system must use a high-level query language which provides support for data stream oriented primitives and operators.

**Rule #3: Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)**

The system must provide mechanisms to compensate for decreased reliability resulting from imperfections of data streams. If an operator requires data from multiple sources it needs to take into account that data will arrive late or not at all and also out-of-sequence. Processing of the data stream must not be blocked in the process.
CHAPTER 2. PRELIMINARIES

Rule #4: Generate predictable and repeatable outcomes The system needs to produce predictable and repeatable outcomes. Especially with regard to replication and graceful degradation it is important that operators of the application proceed purposefully and generate reproducible outcome. . . .

Rule #5: Integrate Stored and Streaming Data It is often necessary to establish context by comparing historical data to the current data stream. Applications which are designed to react to the occurrence of particular states, for example, need to be capable of recognizing said states by e.g. recognizing patterns in the data stream. For this reason the system must be capable of saving the state and combining data streams with historical data while processing.

Rule #6: Guarantee Data Safety and Availability The system needs to provide special mechanisms which ensure high availability. On failure, as time is a fundamental factor in most areas where stream processing systems are used, it is not possible to wait for the system or parts of the system to restart. Instead, fail-over mechanisms are required which will use replication and regular status updates to guarantee that faulty application components will be recognized and replaced in a minimum of time.

Rule #7: Partition and Scale Applications Automatically The system needs to include the possibility to distribute data stream processing to multiple processors and additional nodes in order to ensure high scalability. Data and work load distribution should be completely automated and transparent to users of the application. This requirement is of utmost importance with regard to sustainability.

Rule #8: Process and Respond Instantaneously The system must be able to process high-volumes of streaming data. For this reason all components need to be highly-optimized and the ratio of overhead must be minimized.

While some of these requirements, such as reducing processing latencies by avoiding costly storage operations, are of utmost importance, other are (more) negligible. An example would be supporting a stream query language, which is not necessary for a system to function. If included in a SPS, it increases usability, hence may expand the user base of the system.

2.5 Running Example: Linear Road Benchmark

The LRB was designed by the developers of Aurora and STREAM for comparing and analyzing stream processing systems. Other benchmark proposals exist, such as StreamBench [Lin07] or NEXMark [TTPM02] but none of these seemed to be finished or have been taken up by other researchers. There are other frequently used benchmarks which include querying linked data streams such as SR-Bench [ZDCC12] and LSBench [LP+12] but these perform primarily functional tests in order to determine features like inference expressiveness and are thus restricted to Resource Description Framework (RDF) stream engines[ABV].

The LRB describes a variable toll system for expressways in the fictional Linear City. In this scenario tolls are calculated based on traffic congestion and accident occurrences to regulate traffic. To discourage drivers from using the expressway, tolls become more expensive the higher the traffic volume gets. To accomplish variable tolling, every vehicle is equipped with a sensor that periodically emits its position to a central system and occasionally requests account balance and expenditure information. The system gathers all position reports received to generate statistics,
computes tolls in real-time and transmit tolls back to the vehicle.

Linear City is comprised of $L$ expressways numbered, 0,...,$L$-1 with traffic flowing in both directions. Every expressway is divided into 100 one mile segments each with 5280 position points and has 5 lanes per direction, two of which are used for entering and exiting the expressway. Vehicles using an expressway transmit their positions and current speed every 30 seconds.

The application must implement the following four queries. The benchmark specifies a fifth query (travel time estimation) but since none of the previously published implementations included this query, it will be omitted in this work, too.

- **Accident Detection**
  The system must be able to detect and keep track of accidents. Accidents occur if two vehicles are stopped at the same position for more than four consecutive position reports. Every vehicle entering a segment within the proximity of this accident has to be notified of that accident in order to be able to leave the expressway and thus avoid congestion. An accident is assumed to be cleared if a position report of an involved vehicle indicates that it has been moving.

- **Toll Processing**
  Tolls are levied whenever a vehicle is driving on a congested segment. A segment is considered congested if the average speed of all vehicles in that segment is less than 40 MPH and the number of vehicles in that segment is greater than 50, all calculated within the last 5 minutes preceding the current minute. To compute the latest average speed (LAV), the average speed of each vehicle in each segment is calculated every minute and then aggregated. The actual amount of the toll is calculated based on the number of vehicles or zero if an accident was detected up to 5 segments downstream. Every time a vehicle enters a new segment, the system has to calculate the toll and convey it back to the vehicle. If the vehicle crosses to the next segment the toll is charged. If it decides to exit the expressway no toll will be charged.

- **Account balance**
  Vehicles can make balance enquiries. For the calculation the system has to take into account all tolls charged during simulation.

- **Daily Expenditure**
  Vehicles can request the sum of tolls spent on a given expressway of any day from within the last ten weeks.

The LRB requires the system to process this set of queries with correct results and with a hard response latency requirement of 5 seconds for toll, account and accident notifications, and 10 seconds for daily expenditure and travel time queries.

The input of this application is the simulated traffic of the city over a period of 3 hours, and statistical data for every car, each day and expressway over the past 10 weeks.

Analysis using Linear Road Benchmark is based on the following criteria:

- **Response time:** A hard real-time response latency is given for each query.

- **Accuracy:** Due to the fact that the outcome depends on several parameters, e.g. input order of data tuples, multiple responses are defined as being correct for each query. The results are validated.
• **Supported Query load:** The amount of input data is scaled with the number of expressways. Based on these criteria the so-called *L-rating* is defined as the maximum number of expressways whose data can be processed by the implementing system while satisfying aforementioned requirements. Hence, the benchmark outcome for a system is the number of expressways that it can process. The exact specification can be found in [Ara+04a]. More details running the benchmark and evaluation criteria will be described in Chapter 5.
Chapter 3

Background Concepts and Related Systems

This chapter introduces various concepts and systems for processing large scale data and data streams. In conclusion, I will provide a summarizing review of models and systems related to batch and stream processing.

3.1 Parallel Batch Processing

The enormous growth of the Internet, broadband fiber and mobile networked computing as well as the advent of mobile technologies, social networks and advances in sensor technologies increased the demand for decentralized processing of huge unstructured amounts of data. Traditionally data processing and management were centralized, highly structured and limited in volume. Cost and overhead of organizing mostly unstructured or semi-structured data into relational databases is high. To meet the challenges of processing data at this scale, multiple technologies such as NoSQL, distributed file systems and distributed databases have emerged to overcome limitations of traditional approaches like relational databases. The most notable technology, which is used in the area of Big Data is MapReduce, which will be introduced in this section.

3.1.1 MapReduce

MapReduce [DG08] is a programming model for distributed parallel batch processing of large scale data, as well as a framework that supports the model. It was developed at Google mainly to achieve high performances when indexing web content on a large cluster of commodity machines. MapReduce drew its inspiration from functional programming languages like Lisp. Functional languages have typical features like higher order functions and processing mechanisms that became the basis for the MapReduce paradigm [Kim07]. One of which are so called pure function which have no side effects, are deterministic, stateless and therefore referentially transparent. This means in an expression, every sub-expression can be replaced by a sub-expression of equal value. An expression depends only on its sub-expressions. For instance: Given an expression $T = P(x) + P(x)$ and if $x = y$, then the expression $T = P(x) + P(y)$ holds too. The order of evaluation of each sub-expression is irrelevant. If the function is without side effects, there are no dependencies or influences from outside which enables parallel evaluation.
Programming Model

The user specifies two functions: \textit{map} and \textit{reduce}. The total amount of data to be processed is divided into multiple partitions. The former function takes a partition of the input data and outputs a set of intermediate key/value pairs, where keys are not necessarily unique. The \textit{map} function is called for every partition of the input independently (data interdependency) and thus can be computed in parallel. After all intermediate results have been computed, values that have the same key are grouped together. The \textit{reduce} function takes all values with the same intermediate key and forms an overall result by merging the intermediate values. The latter can also be run in parallel because the output keys are constrained to be working on separate data sets.

The model is best illustrated with an example: The Linear Road Benchmark provides a toll history file which consists of tuples of the form 
\[(\text{vid}, \text{day}, \text{xway}, \text{tolls})\] describing the amount \textit{(toll)} charged for that day \textit{(day)} on the given expressway \textit{(xway)} to that vehicle \textit{(vid)} every day in the past 10 weeks. We would like to compute the sum and the average toll charged to every vehicle in the last week. Figure 3.1 shows the MapReduce program expressed in pseudo-code for computing the sum and average.

\textbf{map}(\text{String key, String val}): \\
// key: file name \\
// val: file contents \\
\text{for each line in val:} \\
\quad \text{tuple = split(line,(,));} \\
\quad \text{if(tuple[1] > 61)} \\
\quad \quad \text{emit(tuple[0],tuple[3]);}

\textbf{reduce}(\text{String vid, Iterator toll_list}): \\
// key: a vid \\
// values: a list of tolls \\
\text{int sum = 0; int cnt = 0; int avg = 0;} \\
\text{foreach (toll : toll_list):} \\
\quad \text{sum += toll; cnt += 1;} \\
\quad \text{int avg = sum / cnt;} \\
\quad \text{emit(vid, avg, sum);} \\

Figure 3.1: A MapReduce example program that computes sum and average of tolls.

The MapReduce framework will execute this program as follows: First the input will be partitioned into chunks. Each chunk gets processed by its own map task. The mapper (as an instance of the map function is referred to) receives an input key - the file name - and the corresponding value - the contents of the chunk. The mapper reads the line checks the day, if the day is greater than 61 (days are represented by a number between 0-69) it will emit the vehicle id (vid) as the intermediate key and the toll s the intermediate value. The result of the mapper is written to disk. After every map process is finished, the output is read and sorted in a distributed fashion. In this so-called \textit{shuffle-phase} the map output is grouped together such that all tolls of a vid from all of the chunks get passed to the same reduce task. The reducer is handed a vid and iterates over all of the corresponding tolls, computes the sum of the list of tolls as well as the average and generates an tuple containing the results for each vehicle, which is written back to the distributed file system.

The whole data flow is depicted in Figure 3.2. The dotted boxes indicate nodes, the black arrows show data transfers on a node, and the purple arrows show data transfers between nodes.

An optimization in this process is to interpose a \textit{combiner} function which is effectively a mini-reducer: it takes all of the intermediate values for a given key and merges them but it only operates on values local to one map process prior to writing the intermediate results. The \textit{combiner} can always be used if the given problem in the reducer function is both commutative and associative. This significantly increases performance since less key/value-pairs are sent over the network.
3.1. PARALLEL BATCH PROCESSING

The MapReduce framework is responsible for data partitioning, load balancing, failure handling and writing out results to the distributed data storage. It was designed to work hand in hand with Google’s own file system the Google File System (GFS) [GGL03], which is a highly scalable distributed file system (DFS) and runs on top of the existing file system in the cluster. It was mainly designed to support applications used and developed at Google thus is highly optimized for their specific needs, e.g. handling a few large files as opposed to handling many small files or streaming reads of files rather than random access. The architecture corresponds to a master/slave pattern. The masterserver is responsible for keeping all metadata, noticing machine failures and coordinating access to the individual files whereas the so called chunkservers hold the actual data. GFS stores files divided into fixed size chunks (as a large piece of a file is referred to as), replicated across multiple nodes at least three times and preferably throughout different racks. The MapReduce framework also corresponds to a shared-nothing master/slave-architecture. The master is responsible for distributing tasks among the workers (slaves), keeping state of running tasks as well as the locations of intermediate and output files and worker health. Workers are being assigned either map or reduce tasks. The framework makes use of the redundancy given in GFS: To minimize the amount of traffic actually used MapReduce tries to distributes the the processing tasks to the data not the other way around (locality).

Fault Tolerance

One of MapReduce’s strength is its fault tolerance mechanism. Since working on top of commodity hardware there is a good chance of machines failing. Even with a Mean Time Between Failure (MTBF) of one year there is statistically a chance of a machine failing every three days. Hence fault tolerance was a major design issue within the MapReduce framework. The functional nature of map and reduce allows for deterministic re-execution of computations: If a worker fails, which is verified by periodically pinging, any map or reduce task in progress is reassigned to different machines and recomputed. Since there is no guarantee that all of the intermediate data computed by the failed worker has been spread out to the reducing machines, the in-progress as well as

Figure 3.2: An exemplary MapReduce Data Flow.
finished map tasks are re-executed. This is not necessary with reduce tasks since reducers are only
going to claim their completion after its results have verifiable been written to the distributed file
system hence been backed up to several different locations.
If a particular map task (or rather its set of keys and values) is causing a crash on multiple machines
MapReduce is eventually giving up on that piece of data and allows the partial computation to
complete with a warning message.
In case of the master failing the MapReduce computation fails.

Limitations
There has been some debate (and criticism) around MapReduce [DS08; Aba09; Pav+09; Lin13;
Sto+10]. Most of which can be attributed to the fact, that MapReduce is often compared to a
relational database management system (DBMS). However, the MapReduce programming model
has its limitations: It is clearly restricted to a certain class of computational problems. Problems
that need to share a global state, which many algorithms, e.g. in machine learning depend
on, are difficult or not to implement since the shuffle phase is the only opportunity for global
synchronization. The data flow is very fix: performing common operations like joins, recursive
and iterative computations requires the need for complicated workarounds. Moreover MapReduce
processes in batches meaning it can only process a fixed size set of data which is already available.
This can be problematic e.g. in the context of processing log files: when the Mapreduce job is
finished, the data is not up-to-date anymore and the process has to be repeated.
Despite the ability to process large data efficiently it is not suitable for more recent demands like
processing data in real-time and processing of data streams. MapReduce is open for improvement
regarding performance: the results of the experimental evaluation in the original paper [DG08]
show that a MapReduce job has a long start-up time to get to peak performance and the peak
performance itself can also be improved as e.g. reading tasks in a map task job are actually four
to six times slower than how fast a disk could actually be read([Aba09]). To overcome these
shortcomings new solutions were created, some of which will be presented in the following sections.

3.1.2 Apache Hadoop
Apache Hadoop [Hadb] is the most popular open source implementation of MapReduce. It was
developed by Doug Cutting and Mike Cafarella in 2004 and used in Nutch, an open source web
search engine. Cutting later joined Yahoo! where it was split of Nutch and further developed.
Scalability being the main goal: Yahoo announced in 2008 to have used Hadoop on a 10,000
core Linux Cluster. Being already an open source project but having now Yahoo! resources it
soon became its own top-level project at Apache. Many companies like Facebook and the New
York Times are using Hadoop. By now many related projects that fall under the umbrella of
infrastructure for distributed computing and large-scale data processing are part of the so called
Hadoop-Ecosystem [Whi09].
Hadoop up to version 2 is an almost exact implementation of Google’s MapReduce with minor
differences regarding the distributed file system and naming conventions. It consists of the two
major components: the Hadoop Distributed File Sysem (HDFS) (though other file systems are
supported) and the MapReduce component itself.
A classic working Hadoop cluster consists of 5 different types of nodes (which are not necessarily
physically different): Regarding HDFS: the NameNode equivalent to the master in GFS which
distributes files among to the DataNodes and keeps metadata and a secondary NameNode which
is in contrary to what the name might imply not a backup NameNode but is rather responsible
for the checkpoint mechanism utilized in HDFS. The secondary NameNode is maintaining copies
3.1. PARALLEL BATCH PROCESSING

of the NameNode’s metadata and logs. Regarding MapReduce: There is the *jobtracker* like the master in Google MapReduce, coordinating MapReduce jobs. A MapReduce job is a submitted MapReduce program and is divided into map and reduce tasks which are scheduled to run on *tasktrackers*, the equivalent to workers. This implementation has some limitations: For one it supports only a maximum cluster of 400 nodes and a maximum of 40,000 concurrent tasks. The NameNode is a single point of failure. Its failure kills all queued an running jobs, which then have to be resubmitted by the user. Also, client and cluster have to be of the same version and applications and work flow cannot migrate to different clusters. Furthermore, since MapReduce itself is restricted to certain computational problems, the lack of support for alternate paradigms such as iterative processing is adversely.

Hadoop 2 was released with major improvements wrt scalability, performance and interoperability with other projects. The most important alteration was to split up the two major functions of the JobTracker, the cluster resource management and the application life-cycle management. A new component is now *Yet Another Resource Negotiator* (YARN). YARN is a framework to develop and execute distributed processing applications. Key components of YARN are the resource manager and an application master. It is able to scale up to 10,000 nodes and most importantly it enables easy integration of other distributed processing systems. The Hadoop 2 is also often referred to as (Hadoop) YARN.

Part of Hadoop 2 is an automatic failover controller, the ZKFailoverController (ZKFC) which periodically checks the health of the NameNode as well as monitors and manages the state of the NameNode [Hada]. Hadoop’s failover mechanism relies on Zookeeper (ZK).

**Zookeeper** [Zoo] is used for distributed applications as a centralized service for maintaining configuration information and provides distributed synchronization and group services. It serves as the high availability backend. Since Zookeeper is used as a component of many distributed systems, I will briefly introduce Zookeeper here.

Zookeeper appears to the developer as a single service. In the background, however, it forms a cluster of individual ZK server instances, called *Zookeeper ensemble*. This ensemble is self-organized and always chooses a *leader* node, which is responsible for synchronization and maintains consistency. Remaining ZK servers are called *followers*. In case of the leader failing, a new leader from the set of existing followers is called within a few moments. The ZK server instances provide a file system-like data structure in which so-called *znodes* can be stored. Znodes are hierarchically arranged data containers and can be addressed within a ZK server instance much like in the way of a standard file system. An important limitation is the limited size of 1MB for data within a znode which is far more than required for coordination in most cases. Zookeeper offers two kinds of znodes: *persistent* and *ephemeral znodes*. Latter are bound to the lifecycle of a client session and are automatically deleted once the session is terminated. Whether a session is still active or not, is detected by the basis of a *heartbeat protocol*. Participants of a heartbeat protocol periodically exchange simple messages (heartbeats), to inform each other about their liveness. If an expected heartbeat is not received within a specific period of time, it is assumed that the respective process has failed [AM10].

Zookeeper’s session management is often utilized by distributed systems to monitor the health of components as does Hadoop where the ZKFC holds a session open for a healthy NameNode. The NameNode is regarded as healthy as long as it responds to periodical pings by the ZKFC. If the local NameNode is active, it also holds a so-called “Lock” znode which is stored as an ephemeral node. In case of the session expiring, the lock node will be automatically deleted [Hada], notifying the standby NameNode that a failover should be triggered.
CHAPTER 3. BACKGROUND CONCEPTS AND RELATED SYSTEMS

Extensions of Hadoop

With the growing popularity of Hadoop, the need to program the processing jobs in Java became quickly another limitation for its use. This requires for one good developer knowledge and a deep understanding of the MapReduce programming model.

As already discussed MapReduce has some limitations. Common tasks in data processing like joins, iterative processing, creating indices are not as straightforward with MapReduce as they are with different systems like database systems (DBS). A typical example of such is the implementation of an inner join, as known from relational DBS and a common task when querying data. What can be easily expressed with the well known Structured Query Language (SQL), must be written in a MapReduce program through various lines of Java code. This opens up the application for errors and inefficiency because the developer is entirely responsible for optimization (which of course also can be an advantage for an experienced developer). In order to lower the threshold for the adaptation of Hadoop, it was necessary to reduce the complexity of creating a MapReduce job significantly [Bit]. The writers of [SLF13] give a good overview of extensions for Hadoop. Among others, they provide the following:

Apache Pig is an abstraction layer on top of MapReduce with a procedural approach. It was originally developed in 2006 at Yahoo! to allow users without programming experience to use Hadoop. To ease application areas like data integration and manipulation it offers Pig Latin, a dataflow programming language that can describe processing pipelines, which are then compiled into MapReduce jobs by the Pig Framework. These are then executed in the usual form by Hadoop. Pig simplifies the use of MapReduce but also prevents the fine granular control of jobs.

Apache Hive is also an abstraction layer for the MapReduce framework which uses a declarative approach. Hive includes a SQL-like language HiveQL, which enables operations like aggregation, joins, counts and group by’s on data sets persisted on in HDFS. As with Pig Latin HiveQL is compiled to MapReduce jobs and executed as those. The advantages and disadvantages of working with Hive resemble those with Pig: With the use of a fairly intuitive scripting language it is possible to process very large quantities of data persisted in HDFS but the disadvantages are more noticeable as in Pig, as processing pipelines get more complex with the use of nested queries where intermediate results have to be temporarily saved. This can lead to high latencies.

3.1.3 MapReduce-like Systems

This section briefly introduces Spark [Zah+11] and Stratosphere [Stra] two systems which have MapReduce-like programming models with extended functionality.

Spark

Spark [Zah+11] is a project from University of California which aims at a weak spot of batch processing frameworks like MapReduce: data reuse and performance. Data reuse is an important characteristic for many iterative and interactive problems. To achieve this, the concept of resilient distributed datasets (RDD) is introduced. RDD is an abstraction for fault-tolerant, parallel data structures and allows users to explicitly persist intermediate results in memory. Keeping data in memory for reuse is a major performance boost for iterative problems. In addition, RDD implements fault tolerance by logging the transformations by which the data was derived instead of the actual data. It is ensured that each record can be restored by lineage, thus saving the overhead of
3.1. PARALLEL BATCH PROCESSING

checkpointing.

An RDD is a partitioned collection of records. Since RDDs are read-only, ensuring consistency is trivial. RDDs are created by coarse-grained transformations through functions like `map`, `flatMap`, which maps to one or more outputs, `sample` etc. Users can indicate which RDD to persist (e.g. memory, disc only) and control partitioning based on keys.

The authors of [Zah+11] show that RDDs can efficiently express programming models from multiple frameworks including MapReduce, SQL, Pregel [Mal+10] or DryadLINQ [Yu+08]. The functionality of MapReduce, for example can be implemented in Spark by the three transformations `flatMap`, `groupByKey` and `reduceByKey`.

Spark showed to be up to 20x faster than Hadoop for iterative applications, 40x faster than real-world data analytics and scans a 1 TB dataset with 5-7s latency [Zah+11].

Stratosphere

Stratosphere [Stra] is the result of a research project at the Technical University of Berlin, Humboldt University of Berlin and the Hasso Plattner Institute in Potsdam funded by the Deutsche Forscherungsgemeinschaft (DFG). It is a highly scalable system, that allows easy creation of complex analytical processing of large scaled data through automatic parallelization, optimization and hardware adaptation. With these automatic optimizations it is possible for users without experience in developing parallel and distributed applications to use the system efficiently [Bit]. Furthermore Stratosphere provides interfaces for relational databases and graph analysis.

The programming model applied in Stratosphere uses so-called Parallelization Contracts (PACT) and extends the MapReduce model by introducing further functions to be able to model complex algorithms required for e.g. machine learning, graph or text mining. A PACT consists of a so-called Input Contract and an optional Output Contract. The former takes a first-order function and input data as an argument. A first order function can be the known map or reduce function or one of the additional functions provided by Stratosphere, e.g. `cross`, which builds a Cartesian product over multiple input sets. The optional Output Contract is used to tell the system how the user-defined code of the first-order function behaves. For example, the `SameKey` contract combined with a map function, specifies that the code of the map function will not alter the key of the input data, hence the type and value of the output key will remain the same. An optimizer uses these hints when compiling the program to a query plan, to improve the runtime [Ale+11]. In contrast to Hadoop, Stratosphere offers the possibility to describe scalable iterative algorithms, which can automatically be parallelized [Stra].
CHAPTER 3. BACKGROUND CONCEPTS AND RELATED SYSTEMS

3.2 Data Stream Processing

This section deals with the basics of data stream processing. First the characteristics of data streams and different data stream models are outlined and the processing of data streams in general is introduced. Subsequently, an overview of two prominent representatives of Stream Processing Systems (SPS), will be given based on their original publications.

Many different technologies such as Active Databases, Publish/Subscribe Systems, Event Stream Processing (ESP) and Complex Event Processing Systems (CEP) played a role in the development of current SPS [AGT13]. Because of the historical background and the different types of analysis, such as monitoring, signal processing, data mining required by stream processing applications, various research communities have contributed to this field. For this reason the terminology used is inconsistent. Whereas, for example, some authors use CEP and Stream Processing interchangeably others stress the differences between EP, CEP and DSMS. Cugola et al. [CM12] propose a generalization and refer to processing of data streams as Information Flow Processing (IFP).\(^1\) They distinguish between two main models in the area of stream processing: the data stream processing model and the complex event processing model. The former has its roots in the database community and systems based on this model - Data Stream Management Systems (DSMS) are derived from Database Management Systems (DBMS). The latter was developed in the same period. The researchers involved had different backgrounds, however. In the complex event processing model not generic data, but event notifications are processed to identify interesting situations [CM12]. I will not elaborate this model any further as this work is focused on processing data streams in general. A feature which all SPS have in common is that they process data before storing them so as to either reduce the amount of data to hold on to or even only to save information gained from processing. This is also referred to processing data online.

3.2.1 Data Stream Model

Informally a data stream can be described as a continuous, potentially unbounded or infinite sequence of data items. The notion of a data stream item differs from relational tuples over events (for CEP systems), xml documents or more complex objects but is always the same for every item in the stream.\(^2\)

The order of the data items is given either implicitly by arrival time or explicitly by timestamp (given by the source of the stream). In the latter case it is possible that due to transmission latencies, data items may arrive in disorder. The stream’s content can only be accessed sequentially in the order in which they arrive. Thus, in order to be able to revisit a data item it must be stored explicitly. Data stream items are usually produced by other applications or external sources, which is why the input rate of stream elements can be unpredictable and changes over time [GO10; CJ09]. The data stream model introduces new algorithmic challenges. Mainly because algorithms generally assume random access to their input. Additionally space required for processing is limited as opposed to the stream itself. This opened interesting research topics such as techniques for trading accuracy for space requirements [Ara06].

In the stream model, data is usually seen as being pushed to the processing system in contrast to e.g. database systems which assume a pull-based model of data access. This inversion is also reflected in the processing of queries: in traditional database systems, the user is the active party. When initiating a query the user receives a result, whereas in stream processing, queries are

\(^1\)Even though this section is partly based on this work I will continue to use the term stream processing since a stream may consist of signals (e.g. sensor data). Therefore information has to be gained from processing first.

\(^2\)I will use tuples and data items interchangeably.
submitted to the processing system once and results are continuously emitted to the waiting user or consuming application [Che03].

### 3.2.2 Continuous Queries

Queries may be generally divided into two types: There are one-time queries, which are executed (as in traditional database systems) once over a snapshot of data, and continuous queries, which are once defined and evaluated repeatedly for an indeterminate period of time over continuously incoming data, producing continuous output. Another distinction can be made between ad-hoc and predefined queries with the latter being known to the system before the data arrives and the former being issued after the data arrives [BBDMW02].

Continuous queries are typically viewed as a directed acyclic graph (DAG) composed of operators as vertices which are connected by directed edges representing a data stream. Operators are computational units that process items from incoming streams and potentially output data items to outgoing streams. These can be algebraic operations or other custom data manipulations defined by user-defined functions (UDFs). There are two categories, into which operators can be divided: stateless operators, where data is processed independently of previously incoming data, e.g. projection, selection or union and stateful operators, where processing depends on subsequent data such as join and aggregate and thus requires the use of storage structures to maintain state.

It is not possible or feasible for the operator to wait until all data items have arrived to execute a join if the data stream is potentially unbounded. This would result in not executing the query at all. These so called blocking operators need to be modified in the context of stream processing since the resources of a system (as opposed to data streams) are limited. To overcome the blocking aspect and because usually more recent data items are of interest most models use the concept of windows [CM12].

**Windows** may be based on:

- *Ordering attributes*: If an attribute exists that defines the order of stream items, e.g. time, a window length can be defined in units of this attribute, e.g. '4 minutes';

- *Counts (tuples)*: A window size can be expressed based on item count.

- *Explicit markers*: Special markers, so-called punctuations, identify the beginning and end of a window.

Whereas the first and third criteria enable windows with variable sizes (number of items), count-based windows are of a fixed size. They all have their disadvantages or can be problematic, e.g. if the input rate is erratic a count-based window can block the processing of the stream, because processing is waiting for the window to fill up, whereas with e.g. time-based windows, the size of the window can be unpredictable. The previous criteria refer to the content of a window, further aspects are used to classify a window: First the direction of movement, i.e. does the window have fixed or sliding endpoints and second the frequency of movement, that is how often are the contents of a window refreshed, e.g. as soon as a new data item arrives or every other time tick [GO10].

Most common is the use of **sliding windows** as well as **tumbling windows**. Tumbling windows store incoming tuples until a limit is reached, e.g. the window is full or after a given period of time if it is time-based. After the limit is reached the window is emptied and filled again. As opposed to a sliding window where the arrival of a new item invokes the eviction of the oldest one in a count-based window, in a time-based window the items are evicted if they do not satisfy the window conditions which are checked as defined in the frequency of movement policy.

A SPS typically provides a set of operators for creating queries. These can be grouped into three
categories: stream relational, utility and adapter ([AGT13]). Stream relational operators are similar to their pendant in relational algebra but adapted to a continuous and non-blocking version. Utility operators are data manipulation tools often needed when dealing with continuous data streams whereas, as the name implies, adapters are operators to connect and convert data from an external source to the format suitable for processing as well as convert and connect the processed stream for consumption by another application subsequent to the SPS.

There are essentially two classes of languages for modeling queries: transforming languages and detecting or pattern-based languages [AGT13]. The former use operators to process input and produce output. This is done either by using declarative languages, which define the expected result of a query and are typically based on the well-known Structured Query Language (SQL) used by databases such as CQL or StreaQuel. Or by using an imperative language, which lets users specify a plan on how to execute the query by combining operators which process the data stream, e.g. Aurora. This is often done by means of a graphical interface, a feature which is provided in Aurora for example. Detecting or pattern-based languages define conditions (usually as patterns) under which separately defined actions are taken. This is commonly used in CEP systems. Existing SPS such as IBM® Infosphere often combine elements from different language classes [AGT13]. In most SPS after a query are compiled in a possible optimal execution plan prior to deployment. The difficulty in processing of data streams results in allocating the limited system resource. As previously stated because of the potential infinity of a data stream, the resource requirements for processing can be unlimited, which makes an optimal allocation of resources extremely difficult. Often, due to limited memory, approximate query answering techniques such as sampling are used. Other systems incorporate load-shedding techniques to overcome resource limitations. Data items are automatically dropped when the input rate becomes too high for the processing capabilities of the system [CM12]. Which items are dropped depends on the policy provided by the system. Aurora, for example, uses Quality of Service (QoS) information, such as the amount of latency tolerated, to identify which data items to discard while processing [Aba+03]. Most systems process data stream items immediately after arriving, one item at a time. Another approach is to process streams in small batches. Latter accumulates items for some duration and computes the query against all data elements at once. Processing windowed operations can take advantage of batched processing. However there is a trade off between processing latency and throughput [LLLWS06]. Processing a stream as a series of so called micro-batches increases throughput but adds overhead for each batch. Latencies of individual tuples being processed will increase which may not be adequate for real-time processing. On the other hand it can reduce resource usage, e.g. regarding external operations such as updating a database. The micro-batching approach is, e.g. used in Spark Streaming [Spa].

3.2.3 STREAM

One of the first general-purpose DSMS is Stanford stREam datA Manager (STREAM) [Ara+04b]. It was developed at Stanford University as a general-purpose DSMS to process high-rate streams and to support thousands of continuous queries [Mot+03]. It introduces CQL, an extended version of SQL with transformation rules to allow querying of both data streams and stored relations. STREAM distinguishes between relations and data streams. A data stream $S$ is modeled as an unbounded bag of $(s, \tau)$-pairs, where $s$ is a tuple and $\tau \in \Gamma$ is a timestamp. A relation is a time-varying bag of tuples and is denoted by $R(\tau)$ for all tuples at time $\tau$. For processing, three classes of operators exist:

- Relation-to-relation operators are derived from well-known SQL constructs and are used to actually perform the processing and transformation in STREAM.
Stream-to-relation operators periodically produce a relation from a stream and are based on the concept of a sliding window. Sliding windows can be tuple-based, containing a fixed number of tuples, time-based, containing tuples in a given time span, and partitioned, containing a fixed number of tuples based on attributes (similar to a group by).

Relation-to-stream operators are used to produce streams from changes to a relation: \texttt{Istream(R)} contains a stream tuple, whenever a tuple is inserted into \texttt{R}. \texttt{Dstream(R)} contains a tuple for every tuple which is deleted in \texttt{R} and \texttt{Rstream(R)} contains all tuples contained in \texttt{R} at every time instant.

To illustrate the syntax let us consider the following problem in the LRB: we need to know which vehicles are involved in an accident. A vehicle which transmitted four consecutive position reports with a velocity of zero is considered an accident vehicle. Position reports are emitted every 30 seconds. Therefore we need a window of 4 * 30 seconds.

\begin{verbatim}
Select Istream(distinct(vid))
From PosReports [2 minutes] Where spd = 0.
\end{verbatim}

In this query the stream-to-relation operator time-based window is used to create a relation which is then processed by the relation-to-relation operator’s projection, duplicate eliminator and filter operation. The output of the latter is then converted into a stream by using the relation-to-stream operator \texttt{Istream}.

After specifying a continuous query and registering it with the system a query plan is compiled. Query plans are comprised of operators, queues to buffer references to tuples moving between operators and synopses to store operator state. Figure 3.3 shows a query plan for the following query adopted from [Ara+04b]:

\begin{verbatim}
Select * From PosReports [Rows 1000], AccDetection [Range 1 Minutes]
Where PosReports.segment = AccDetection.Segment And S1.A > 10
\end{verbatim}

A select, a binary-join, and 2 window operator, one count-based, the other time-based are used in the exemplary plan. Operators are connected by queues: \texttt{q1} and \texttt{q2} hold the tuples from the incoming stream, whereas \texttt{q3} and \texttt{q4} each contain the output of the preceding window-operators. Tuples derived from the joined relation are contained in \texttt{q5}, and all tuples passing the select operator are contained in \texttt{q6}. Since the select operator is stateless it does not have a synopsis as opposed to the rest of the operators shown. The binary join operator maintains a synopsis for each of its inputs to perform a join. This example shows a common phenomenon with the use of synopsis: synopsis 1 and 3 (respectively 2 and 4) materialize an almost identical content. To avoid such redundancy, the concept of synopsis-sharing was introduced. Two similar synopses are replaced by lightweight stubs and a single store holding the actual data. Since stubs implement the same interface as synopsis they can be used interchangeably.

STREAM implements an adaptive query processing to dynamically adapt to changes in query load. So as to be able to accomplish this STREAM has a component called StreamMon which is capable of reorganizing and optimizing running queries.

### 3.2.4 Aurora

Aurora [Aba+03; Che03] is another example of a research-based DSMS. It was developed in collaboration between Brandeis University, Brown University and MIT. As opposed to STREAM, which uses a declarative language for defining queries on data streams, Aurora uses the procedural approach: The main task of the system is to handle incoming data streams exactly as defined by the application administrator [Aba+03]. It uses the boxes-and-arrows paradigm which is found
CHAPTER 3. BACKGROUND CONCEPTS AND RELATED SYSTEMS

Figure 3.3: Query plans in STREAM.

in many workflow systems to express continuous queries [RMCZ06]. Queries are assembled in a graphical interface as a DAG with operators as boxes and arrows depicting the directed flow of data stream items. There are eight built-in operators with different functions, such as filter, windowed operators, joins or resample, which generates a partially synthetic stream by interpolating tuples between actual tuples of an input stream.

Aurora also offers a storage layer to maintain historical queries, mainly to support ad hoc queries [Aba+03]. It supports continuous and ad-hoc queries as well as views. In continuous queries data items are processed once, they flow through a network of boxes, in which they are processed until the path ends, after which data items are removed (dropped) from the network without persisting. So-called connection points in a query graph as depicted in Figure 3.4 provide the possibility to modify the network dynamically. At these points new boxes and arrows can be connected to e.g. provide an output to other applications. They also offer the opportunity to temporarily store data persistently to e.g. provide data for historical queries. It is necessary to specify how long the data will be stored. Figure 3.4 shows three different paths each representing a continuous query, an ad hoc query and a view, respectively. The QoS specification at the end of each path are used to control how resources are allocated for processing.

Aurora’s architecture is orchestrated by its scheduler component which determines which operator (box) to run. A storage manager is used to buffer queues if main memory is used. To assure fluent processing of tuples, this is constantly being monitored. Every time Aurora detects overload, load shedding is applied based on the QoS information. In case load shedding is not working it will try to transform the original network to perhaps uncover opportunities for load shedding. Should this not be sufficient enough, it will try to retune the scheduler based on new statistics or by switching scheduler disciplines.
3.2. DATA STREAM PROCESSING

3.2.5 Other Systems

The development of commercial SPS is closely linked to the development of the previously introduced DSMS. For example TIBCO® StreamBase [Strb], a widely used DSMS which incooperates CEP capabilites, is the commercialization of the Aurora project. It’s programming model allows the use of two languages: the visual language EventFlow and the declarative SQL-like language StreamSQL. The latter is text-based and is similar to CQL whereas the former resembles Aurora’s graphical language. While the languages are conceptually different, they can be used almost interchangeably. On runtime applications are deployed on a so-called on a StreamBase server. Applications are organized in containers and it is possible to run multiple containers, hence applications, interconnected by streams on one server. StreamBase utilizes a replicated standby server to provide fault tolerance [AGT13].
3.3 Parallel and Distributed Stream Processing

This chapter will deal more closely with the main challenges that arise from distributing processing of data streams and presents briefly different exemplary systems.

3.3.1 Challenges

Distribution, Parallelization and Partitioning

One approach to distribute processing of continuous queries is to partition query plans and to distribute query operators among nodes. This can either be done prior to deployment or dynamically on runtime, which is called adaptive query plan partitioning. A variant of such is called Eddies. Tuples are routed to the operator with the smallest load [ZR11]. Problematic with this approach are bottleneck operators. To eliminate these bottleneck operators are parallelized and data is partitioned. Partitioning data into multiple subsets, which can be processed by individual processes (and/or processors), is one of the key aspects in parallel data stream processing. Because processing a query often consists of multiple steps with possibly every step requiring its own partitioning strategy, shuffling stages to prepare data partitions for parallel computation are necessary [Zha+12].

Reliable Processing and Messaging

As opposed to batch processing, where reliability is not critical because a failed task can always be re-executed from the beginning, stream processing requires more attention on reliable processing. In a distributed system with a shared-nothing architecture, sharing state depends thoroughly on message passing. It is important to know what data delivery model is given in a system since it can change the semantics of an application e.g. when counting data items in a stream. Reliable messaging requires, especially with regard to fault tolerance (messages have to be tracked) extra overhead, which some systems choose not to add. Systems can offer the following delivery approaches: exactly-once, at-least-once, at-most-once or best-effort [YRP02]. The former assures every message to be processed exactly once whereas the subsequent approach only assures that a message will be processed by replaying messages in case of failures. The latter does not impose a strategy but tries to deliver all messages which can be delayed e.g. due to overload and the sender is not informed of a successful delivery. In stream processing data items flows through a chain of operators until they reach the final consumer. Each data item produces its own directed acyclic graph of descendant data items. If reliable data processing is required one must track each descendant item, this is called lineage tracking.

High Availability and Load Balancing

Due to the temporal dynamic stream processing systems have to cope with changes in load distribution. If load continuously increases the system has to find ways to spread the load. Load balancing is the ability to distribute work load evenly across several components. As the number of participants grows, the risk of a failure especially in the context of commodity hardware increases and must be addressed. The ability to maintain unaffected processing while simultaneous failures occur is called high availability.
Scalability and Fault Tolerance

The difficulty lies in developing systems which provide scalability while remaining fault tolerant. As already stated, in a distributed system the probability of a component failing is high. If a component fails due to some unpredictable event, the system has to provide means to replace the component. A common approach is to utilize stand-by nodes. Error handling in that context is difficult because the state might depend on the computation of all previously processed tuples. Replaying of all tuples from the start is generally not possible hence mechanisms for persisting or replicating state are needed.

3.3.2 Distributed Stream Processing Systems

When looking at systems that offer distributed stream processing, one can observe the following approaches to accomplish distributed stream processing:

1. MapReduce-based systems, which enable stream processing on top of parallel batch processing by using micro-batches and trying to reduce the overhead of MapReduce to make processing faster, e.g. by working in-memory and using micro-batches and allowing pipelined processing to support continuous queries.

2. DSMS-based systems, which distribute the execution of queries and/or operators, exploiting inter-query and inter-operator parallelism.

3. Data parallel SPS which affiliate both approaches.

An example for the first approach is MapReduce Online [Con+10]. The authors of [Con+10] have developed the Hadoop Online Prototype (HOP) based on Hadoop’s architecture. The framework was modified to exploit pipeline parallelism. Instead of waiting for a map or reduce function to complete its processing, mappers push intermediate data over the network to the appropriate reducers which are connected via Transmission Control Protocol (TCP) sockets. Reducers themselves are also able to push its results to a mapper of a subsequent job. This way it is possible to run MapReduce jobs continuously, enabling continuous processing of data streams.

Another example is Spark Streaming [Spa], which extends Spark, introduced in Section 3.1.3, by transform streams into RDDs, so that the stream can be processed as a series of deterministic micro-batch computations.

An example for the second approach is Borealis [Aba+05] and the Stream Processing Core (SPC). Borealis is a second generation DSMS, based on Aurora’s stream processing engine. Queries in Borealis are seen as a network of operators, whose processing is distributed to multiple nodes. SPC is another example of an early distributed SPS. Queries in SPC are described using a data flow graph consisting of Processing Elements (PEs). PEs contain the processing logic and have an input and output port for streams. They can be interconnected by subscribing to another PE’s output stream in a publish/subscribe manner and can be run on different nodes.

Examples for the latter approach are Storm, S4, Samza [Sam] and Muppet [Lam+12]. These belong to a new class of systems exploiting intra-operator parallelism by adapting MapReduce’s approach for partitioning data. Unlike MapReduce these systems process jobs potentially infinite. In contrast to Storm For example SCSQ [ZR11] is a DSMS that parallelizes execution of continuous queries by introducing a parasplit, a stream splitting operator to partition streams of high volumes. Streambase is another example. It is possible to scale out applications by running multiple StreamBase servers. However, the partitioning of workload and data distribution lies in the responsibility of the user. For a single StreamBase Server, the system offers the possibility of partitioning data to run components in a separate threads,
3.4 Summary

This chapter formed the theoretical foundation for the following systems presented in this work. First the MapReduce paradigm and some systems implementing the paradigm have been introduced. Then an introduction of stream processing has been given. In the following section relevant aspects for distributed processing of large scale data were treated and some of the available systems were briefly described. It was established that there are 3 distinct approaches for distributed systems to process large scale data streams: micro-batched processing, inter-operator parallel and (intra-operator) data-parallel stream processing systems.

Figure 3.5: Overview of presented Systems.\(^3\)

Figure 3.5 shows all systems mentioned of this chapter and does not depict an overview of all available systems relevant to this subject. But it shows how the introduced systems overlap and exclude or include certain paradigms like relational or non-relational, stream, batch and distributed processing and how these paradigms are connected to themselves. The systems in blue belong to the group of parallel systems. The description and evaluation of the red-dotted systems are the subject of Chapters 4 and 5.

---

\(^3\)Adapted from [LHB13]
Chapter 4

Case Studies

In this chapter current Storm and S4 will be contrasted. By this means, it will become clear how some of the challenges described in Section 3.3.1 are dealt with in concrete implementations. At first, the procedure for selecting the systems in question will be explained and the set of criteria in use will be defined. Subsequently, each system will be described in detail. Finally, a summary of all details, including a table summarizing the main aspects will be provided.

4.1 Method and Set of Criteria

The selection of systems is an in-depth online research and a review of relevant publications. The systems introduced in this chapter differ from the previously introduced SPS in the sense that they are programming frameworks for developing stream processing applications rather than being a complete system, which offers more user friendly descriptive languages or a graphical user interface for assembling queries such as InfoSphere or StreamBase. In that sense the systems in question are more comparable to the MapReduce Framework with the main difference that computation is run indefinitely. Both frameworks are used to implement parallelized operator-based processing of large data streams. The description of the systems is in line with the following set of criteria:

- Main Concepts
- Architecture
- Programming Model
- Parallelization & Scalability
- Fault tolerance

Information was gathered through online research on the providers’ websites including complimentary publications (where available). This was partly supplemented with a review of available documentation and forum discussions. For the scope of this work, two exemplary systems were chosen for closer evaluation:

- Storm
- Apache S4
Both systems are prominent representatives of distributed stream processing systems that exploit data parallelism. Moreover, they show different approaches of exploiting data parallelism, and have unique architectural designs. Storm is by far the most widely used system and even though performance studies on S4 have been carried out [CCM12] they have not been compared too each other in detail thus were chosen as appropriate candidates for a detailed case study.
4.2 Storm

This section describes the Apache Storm framework. The findings of this section are based on the documentation [Mar12a] if not stated otherwise.

Apache Storm [Sto] is an open-source general-purpose framework for distributed real-time computation. It was originally developed by Nathan Marz in 2011 at Backtype later acquired by Twitter where it is used to process tweets in real-time for their real-time analytics products. As of September 2013 Apache has accepted Storm into its incubator program. It is designed to be scalable, fault-tolerant and robust and guarantees zero data loss. Storm is written in Clojure mostly and partly in Java [Mar11]. Clojure is a modern dialect of Lisp and therefore supports a functional style of programming which, as previously established, greatly simplifies concurrent programming [Jon13].

Storm was benchmarked at processing over a million 100 byte tuples per second per node.\(^4\) Three use cases are suggested by Nathan Marz: stream processing, continuous computation and distributed remote procedure calls.

4.2.1 Main Concepts

Storm views a data stream as an unbounded sequence of tuples. A tuple is an ordered list of named fields, where a field can be an object of any type as long as it is serializable. A Storm cluster can be compared to a Hadoop cluster. Instead of running “MapReduce jobs”, so-called topologies are submitted to the cluster. Whereas a MapReduce job will eventually be completed, a topology is processed forever until explicitly terminated. Topologies consist of spouts and bolts, which contain the processing logic:

A spout is defined as the source of one or more data streams. In a spout the logic for wrapping external sources is written. The core function implemented by a spout is nextTuple where tuples are emitted to the topology. There are different spouts available contributed by the open source community e.g a spout for watching active files/directories, a spouts for consuming messages from a message queues or spouts for watching database transaction logs or table rows.

A bolt represents a processing step. It can subscribe to any number of spouts and bolts and can produce new streams to be processed by subsequent bolts. The core function of a bolt is execute allowing transformations or functions to tuples to be applied. This includes operations like filtering, aggregation or joining streams as well as persisting data to databases or handing to other applications.

A topology is constructed as a directed graph where the nodes are spouts and bolts, connected by streams. It describes the whole application logic. Communication between spouts and bolts is only possible through subscribing to streams. Spouts are not able to consume streams, since they are meant to model and emit streams.

The user submits topologies to the storm cluster where they will be running indefinitely until explicitly terminated by the user.

Figure 4.1 shows a topology excerpt from our running example as a directed graph in which vertices are spouts or bolts connected by streams. The directed edges indicate which bolt subscribes

\(^4\)This fact is based on a benchmark that was completed on a system with the following specifications: 2x Intel E5645@2.4Ghz and 24 GB of main memory (http://storm-project.net/about/scalable.html, last accessed 03/20/2014).
to the output stream of other bolts or spouts. SplitStreamSpout splits the input stream into one stream containing all position reports and another one containing all account balance queries. Former stream is replicated because it is consumed by SegmentStatsBolt, which computes the average speed in each segment of an expressway and the TollNotificationBolt, which emits toll notifications. To calculate the toll amount, depending on the congestion of an expressway, the TollNotificationBolt also subscribes to the stream novLavs where a tuple contains the number of vehicles (nov) and the latest average speed (lav) for one segment. All emitted notification are consumed by a FileWriterBolt which writes all notifications to an output file. This only describes the abstract relationship between spouts and bolts. The actual physical layout of a running topology is controlled by Storm (configurable by the user) as we will see in the following sections.

4.2.2 Architecture

Figure 4.2.2 shows a minimal storm cluster setup. A Storm cluster consists of the following components: one master named Nimbus, Supervisors, running on each worker node, Zookeeper for coordination, and a Storm UI. To describe a working Storm cluster, the following primitives are needed: tasks, executors and worker processes. Tasks which perform the actual processing belong to a bolt or spout. They are assigned by the Nimbus and run by means of a worker process. A worker process (not node) runs its own Java Virtual machine (JVM) and executes a subset of a specific topology. Worker processes may run one or more executors for one or more bolts/spouts. An executor is a thread invoked by a worker process. When setting up a storm cluster the user configures Supervisors to dispose of a number of slots which correspond to the number of possible worker processes of this node. Typically this is the same as the number of cores in a machine and is set to four by default. The number of utilized worker processes is set when defining the topology.

The Nimbus is responsible, among others for accepting submitted topologies and distributing the code among the workers. An important component of the Nimbus is the scheduler. When a topology is started, the scheduler has to assign executors to worker processes and assign worker processes to slots. If worker slots are available, executors are assigned to them. If there are no empty slots they are assigned to workers with the lowest utilization. The default scheduling strategy is done round-robin with the aim of an even allocation ([ABQ13]). If a topology is killed by the user the Nimbus first stops all spouts and waits for a user-specific period of time for the bolts to finish.
4.2. STORM

Figure 4.2: Storm Components.

processing the already consumed tuples.
Other tasks of Nimbus are to monitor the cluster for failed tasks and reassign them. The Nimbus
does not directly control the workers but rather through Supervisors, which start and stop pro-
cesses based on their assignments. When a Supervisor is started it registers itself with the Nimbus
and periodically sends out heartbeats to inform the Nimbus of its state. Supervisors monitor their
worker processes and inform the Nimbus of failed tasks, so that they can be re-assigned to other
workers.
Storm uses a Zookeeper cluster for managing communication between the Nimbus and the Super-
visors.
The Nimbus keeps information about running topologies and tasks and available supervisors on
Zookeeper as do Supervisors.
Storm uses ZeroMQ (OMQ), as a transport layer, which is a fast asynchronous Messaging Library
for distributed and concurrent systems. For serialization Kryo is used. Kryo [Eso14] is a Java
serialization framework with focus on speed, efficiency and easy use.
The Storm user interface (UI) provides an overview of available workers, slots and running topolo-
gies. For every topology statistics and state information of workers are available. It is also possible
to kill, deactivate, start and rebalance a topology from the UI. More commands for starting the
Nimbus or a Supervisor, for example, are available with the Storm command line client.

4.2.3 Programming Model

There is no explicit programming model in Storm. The application is defined in terms of spouts
and bolts, which correspond to user defined operators. The API has interfaces for spouts and bolts,
for which developers essentially have to implement two primary handlers: execute() in bolts
and `nextTuple()` in spouts. Whereas a bolt is implemented as if it processed the complete stream it is subscribed to, the actual bolt task only processes a partition of its subscribed stream (or the whole stream if there is only one task). This is similar to the procedure used by MapReduce where each map and reduce function only processes a partition of the data. But opposed to MapReduce, Storm offers different built-in stream groupings to define which tuples are distributed to which task:

1. **Shuffle grouping** distributes tuples at randomly but equally to each task (round-robin), which is equivalent to data being passed to map tasks in MapReduce.

2. **Fields grouping** partitions the stream by field values specified in the grouping, which corresponds to MapReduce’s built in partitioning for reduce tasks. This is accomplished by taking a hash of the field values and modding it by the number of tasks.

3. **All grouping** replicates tuples across all tasks of a bolt which is sometimes needed when bolts are joining streams.

4. **Global grouping** causes the entire stream to go to the one task of a bolt with the lowest id.\(^5\)

5. **None grouping** is currently behaving like shuffle grouping. A feature is planned to use this grouping to be able to pack multiple bolts/spouts in one thread where none grouped tuples would be distributed to either task of the subscribing bolt.

6. **Direct grouping** is fundamentally different from the other groupings: The emitting component is responsible for distributing the tuple to a task of the subscribing bolt. This can only be done after a stream has been declared as a direct stream, which can be accomplished by using one of the `emitDirect` methods.

7. **Local or shuffle grouping** preferably distributes tuples to tasks in the same worker process if present or acts as shuffle grouping otherwise.

It is also possible to define a custom stream grouping by implementing the `CustomStreamGrouping` interface.

Let us consider the following problem in our running example: we need to detect an accident. An accident is defined, as a vehicle reporting four consecutive position reports with a speed of zero. We have one spout which emits all position reports and one bolt remembering and counting every stopped vehicle. In this programming model we have to implement the bolt as if it processed the whole stream, so we would use a map to keep track of stopped vehicles. To distribute computation, we need to set the parallelism hit. Since we need to know about stopped vehicles per segment in each direction on each expressway, we can partition the data on these fields. If we want to have one instance of the bolt to watch over 10 segments, for example, we would set the parallelism hint to 10. So when adding the bolt to our topology we would use fields grouping on expressway,segment and direction. To know what parallelism hint to set, it is best to test the topology. If the capacity (based on process latency and the number of processed tuples) of a bolt is close to 1 over a window of 10 minutes one should increase the degree of parallelism ([Mar12b]).

### 4.2.4 Parallelism & Scalability

The parallelism in Storm is achieved by partitioning the stream so that multiple instances of the same bolt can process a part of the stream concurrently as described in the previous section.

\(^5\)This can be useful for a bolt computing more than one problem, where one would be having to do a reduction of a specific stream.
4.2. STORM

The developer can define the degree of parallelism by setting a parallelism hint when defining the topology. The parallelism hint defines the number of executors storm will spawn for a bolt or spout (that is how often a bolt/spout is replicated in the cluster). The developer is also able to set a number of tasks for an executor which is one by default. Typically one would want one task per thread. Setting the task number higher would increase the number of instances running for a bolt/spout but will not increase the degree of parallelism because an executor has always one dedicated thread so that tasks are run serially on an executor. The number of executors can be changed by invoking the storm rebalance command whereas the number of tasks of a bolt/spout in a topology remains static ([Nol12]). Adding an executor would result in redistributing the tasks among the new number of executors, which is depicted in Figure 4.2.4. This means in order to be able to rebalance a topology one would need to set more than one task per executor.

Picture 4.2.4 depicts a simple topology and one possible physical layout at runtime before and after rebalancing of the topology. The spout is initially created with a parallelism hint of one and the task number set to two, by invoking the rebalance command the number of executors is increased to two so that each spout executor processes one task. This feature of rebalancing a topology makes Storm capable of scaling out on runtime (elastic scaling).

4.2.5 Fault Tolerance & Reliability

Storm’s fault tolerance mechanisms include the following: Workers in a running topology send heartbeats through Zookeeper to the Nimbus to indicate that they are running. The Nimbus monitors these heartbeats and will tell the Supervisors to kill workers that fail to send heartbeats. Failed workers are then restarted by their Supervisors. If a worker is unable to start or repeatedly fails, the Nimbus will reassign the worker to another node in the cluster. Even though Storm corresponds to a master/slave architecture and could constitute a single point of failure, a failed
Nimbus has little impact. All processes will remain active. If failures occur with workers they will be restarted by their Supervisor but in case a machine fails, which includes the death of the supervisor, or if a worker continuous to fail, the workers cannot be reassigned to other machines by the Nimbus. Therefore a failed Nimbus will need to be restarted. Since all information about state and configuration is kept on Zookeeper and disk, Supervisors and Nimbus are completely stateless and if they run under supervision they will possibly fail, be restarted and recover without data loss. If Zookeeper fails, which is highly unlikely since if deployed correctly it can guarantee high availability, processing will fail, because any configuration and state regarding the cluster is kept on Zookeeper.

Storm ensures that every tuple coming of a spout is processed at-least-once. This is accomplished by tracking the tuple in its DAG. A tuple can be emitted as anchored to the one received or to an ID if it created in a spout. Multiple anchorings on the path down processing form a DAG as shown in Figure 4.4. A spout emits a tuple $a$, which is then processed by bolt $B$ which emits a tuple $b$ and $c$ and so on. All Tuples $b$, $c$, $d$ and $e$ are descendants from tuple $a$. In any succeeding bolt which needs to be included in the DAG, the tuple needs to be anchored as well. If a tuple is emitted as anchored and a failure occurs downstream the root tuple will be replayed. It is possible to anchor a tuple with more than one tuple by using a list of input tuples as an argument when emitting.

To let Storm know that a tuple has been fully processed the `ack` or `fail` method is invoked. This is important because Storm uses memory to track a tuple and will otherwise run out of memory. A tuple is not considered fully processed until all nodes in the DAG have acked the tuple. Unless all tuples in the DAG have been acked within a given time span, which can be configured, it is considered to have failed and the root tuple gets replayed.

Storm uses a designated task called `ackers` to track tuples in its DAG. By default there is exactly one acker but this can be configured for each topology. The tracking itself works as follows: For identification tuples, are assigned a random 64 bit ID on generation. After a spout task has generated a tuple, it sends its task ID to the acker responsible for tracking the tuple. Which acker task is responsible for a tuple is determined by using a hash function. The acker keeps a map for each tuple it tracks with the following information: the task ID that created the tuple and the ack val, a 64 bit number which is the xor value of all tuple IDs that have been created or acked in its DAG. Figure 4.4 shows the DAG for tuple $a$. Its first descendants are $b$ and $c$, hence the ack value, the XOR value of $b$’s and $c$’s message IDs would be 0001. For every new descendant the ack value is xored with the new id. This way the space needed to track a complete DAG is always of fixed size, 64 bit. When a tuple is acked the ack value is xored with the ID again. If a value is xored with itself the result is zero. Thus the acker knows if a DAG is completed, when all descendants have been acked and the ack val is zero. The acker can then inform the originating spout task to dismiss the root tuple.

![Figure 4.4: Tuple tracking in Storm.](image)

This mechanism only ensures continuous processing with an at-least-once semantic because a
replaying of tuples could change the result of a stateful operation. If a tuple is replayed it will be reprocessed from the beginning and will pass all operators again. Therefore, if an operator is not idempotent e.g. when counting occurrences, it will be counted twice. Exactly-once-semantics can be achieved by making all operations idempotent or by using Trident, presented in the following section. With Trident it is also possible to implement stateful stream processing. Trident batches are given transaction IDs (txid). If a batch is replayed it allows for idempotent updates by comparing txids.

4.2.6 Extensions

Trident

Storm provides a high-level abstraction similar to Pig and Hive on top of Storm named Trident which provides for micro-batched processing of data streams. The main advantage in using Trident is, it allows for recoverable stateful stream processing. Moreover instead of the at-least-once message processing guarantee, it processes tuples exactly once. Trident adds more complexity to Storm, thus performance is lowered but it provides a means of persisting the state of operators. The API provides higher-level constructs called *operations* which are bolts underneath and correspond to relational-like operators. It has built-in support for filters, joins, groupings, aggregations and functions. A trident topology is constructed similarly to a Storm topology but instead of adding bolts which subscribe to streams of spouts and bolts, a stream is defined by opening a Trident spout and operations are chained to work on that stream subsequently. Trident batches are constructed in spouts where the size of a batch can be determined. The right batch size depends on incoming throughput and the degree of latency required. The size of one batch is fixed throughout the topology, but the number of partitions in a batch can be changed by applying a grouping operator and changing the parallelism hint. Operations used on TridentTuples only operate on a subset of the fields in a tuple. This projection is so to speak included in the process and is working very fast.

The result of an operation is either the same TridentTuple e.g. when filtering, the same TridentTuple added with new generated fields when using functions, or all fields replaced when using aggregators. If a grouping is used before an operation the output consists of the grouping fields and the result of the operation.

DRPC

Storm offers another type of topology known as Distributed Remote Procedure Call (DRPC), which executes Remote Procedure Calls on demand in a distributed fashion. This enables Storm to execute ad-hoc queries. This is coordinated by a DRPC server which runs as a connector between client and the Storm topology. The developer can issue a query per command line which is then sent to the server where the query is emitted as a tuple into the topology. After the query has been computed, the answer is send back to the client.
CHAPTER 4. CASE STUDIES

4.3 Apache S4

This section describes the Apache S4 framework. The findings of the following sections are based on the original publication supplemented with findings by the authors of [NRNK10; CCM12] and documentation available on their webpage [S413].

Apache S4 [NRNK10] was initially developed at Yahoo!, where it was used for processing developer feedback related to online advertisement placement. The name is an acronym and stands for “Simple Scalable Streaming System”, which mirrors the motivation behind this project: to create a system for fast and easy development of an application for processing unbounded data streams that are easily scaled.

The main requirements were to guarantee high availability while allowing easy scalability using commodity hardware. After being released as open source in November 2010 it became an Apache incubator project a year later.

S4 is an event-driven decentralized system inspired by SPC, introduced in Section 3.3.2 and the MapReduce paradigm as it uses a key-based distribution of data in its processing model. It has a modular design and offers pluggable event serving policies like load shedding and blocking.

The developers of S4 initially made two important assumptions to simplify the initial S4 design: they accept lossy failover and elastic scale out will not be necessary.

S4 was benchmarked at processing over a 200 thousand messages per second per stream.

4.3.1 Main Concepts

In S4 processing logic is encapsulated in apps which consume and produce streams as topologies do in Storm. Multiple apps can be coupled by subscribing to outgoing streams. A stream is defined as a sequence of events with events consisting of tuple-valued keys and several attributes. Making use of so called adapters which are apps themselves, external sources are converted into keyed events, which can then be injected into other apps. Adapters are the counter part to spouts in Storm with the difference that they are decoupled from the app containing the actual application logic.

Events are processed by Processing Elements (PEs). PEs are the equivalent to Bolts in Storm, which are both components which contain the main application logic. PEs are implemented as prototypes by the developer and specify how to process an event. A PE can generate output events which again can be consumed by other PEs. Similarly to reducers in MapReduce, an instance of a PE consumes exactly the events in a stream that correspond to the value on which it is keyed.

So for every unique key in a stream there will be one instance of the consuming PE.

This is illustrated in Figure 4.5, which shows part of the LRB app. The task is to compute the latest five-minute speed average in each segment. The LRB app consumes the output of the adapter app, which consists of a single adapter, that consumes the input stream of the simulation and generates a stream of position report events. In the LRB app an instance of VehicleSpdPE is keyed on a unique vehicle id and receives every PositionReportEv of that vehicle. For example, PE1a consumes every position report for the vehicle with the vid 345. The VehicleSpdPE accumulates the speed of that vehicle for every segment it crosses and outputs the average speed of each segment as soon as the vehicle changes the segment. These VehicleSpeedEvents use the unique segment id as a key, which comprises the expressway number, the segment and the driving direction (xsd).

These events are consumed by a SegmentStatsPE, of which there are exactly L*200 instances.

---

6Traffic generated by queries posted on the search engine was processed to get tuning parameters for the search advertising system.

7No further details on the performed test are available [S413]

8The difference is that a reducer can have multiple keys which its consumes whereas a PE instance only consumes items with the value for the key its keyed on.
one for every segment in each direction. This PE accumulates the speed values of all vehicles driving on the same segment and calculates the average for every minute in that segment.

There is a special class of PEs called *keyless PEs*. They act like mappers and consume all events of the type to which they are subscribed to. These are usually used at the beginning of processing in an app, to convert (map) external stream items into keyed *events*. The initializing of PE objects and stream partitioning by key values is handled by S4.

### 4.3.2 Architecture

S4 implements a decentralized and symmetrical cluster architecture in contrast to Storm where the Nimbus is a central point responsible for coordination. In S4 apps are deployed on logical S4 clusters. An S4 cluster can consist of a potentially unlimited number of partitions, each containing a Processing Node (PN), the logical host to PEs. The architecture as depicted in Figure 4.6 can be divided into the PN core layer and the communication layer. The core layer of a PN has the following components: an *Event Listener* for external events, a *Dispatcher* for routing events and an *Emitter* for outgoing events.

The core component is the Processing Element Container (PEC), which is responsible for delivering events received by the Event Listener to the corresponding PE. The dispatcher component is used by the PEs for sending events to other PNs. Events are routed to PNs, using a hash function of the values of all known keyed attributes in that event. If an event arrives to a PEC with a value for the keyed attribute for which there is no PE instance available the cloning of the PE prototype is triggered. This ensures that every keyed PE belongs to a specific PN, with the exception of *keyless PEs*, which can be instantiated on every PN. This architecture resembles the Actor model, where PEs are actors, supervised by PNs, that communicate asynchronously with each other by sending messages (events) on streams and are able to replicate themselves if necessary. In case a PE in an application consumes events with a large number of keys, it is advisable to implement
mechanisms to remove PEs from the network to reduce the number of PE instances. This can be done by invoking `setPECache`, to add a time-to-live (TTL) to a PE. The TTL can be determined through either a maximum number of instances, in which case PE instance which was not used the longest is removed or by time, so that when an instance has not received an event in specified period of time it will be removed.

Communication between PEs on different PNs, managing mapping of physical node to PNs, automatic failover and load balancing are handled by the communication layer. The communication layer API allows other systems to send input events to nodes in an S4 cluster in a round robin fashion, which are then consumed by adapter apps. The communication layer is based on Zookeeper, where coordination between nodes is handled. Zookeeper assigns S4 cluster tasks to physical nodes. It distinguishes between active nodes and idle nodes. Idle nodes can be used when needed or specifically registered as a standby node for multiple active nodes which may have been assigned a particular task.

### 4.3.3 Programming Model

Apps in S4 are built as a graph of PEs interconnected by event streams. The architecture of S4 enforces the developer to implement the Actor Model. The developer implements the abstract class `App` where remote streams, PE prototypes and events belonging to the app are declared and assembled. Events are simple Java immutable objects, although, as in Storm, immutability is not enforced. Therefore events fields should not be updated and re-emitted but rather cloned. PEs are meant to be generic and reusable through configuration. A PE prototype is identified by the following four characteristics:

1. the *functionality* defined by a PE class and its configuration,
2. the *types of events* which are consumed by the PE and
4.3. APACHE S4

3. the *keyed attribute* which the consumed events entail.

An instance of a PE is characterized by a fourth item:

4. the *value of the keyed attribute* which is consumed by the PE.

A PE can contain state variables just like bolts and spouts in Storm but since PE instances are cloned from the prototype and thus never use the constructor ([S4j]), they must be initialized by implementing the `onCreate()`-method. The primary handler is `onEvent`, which is invoked for each incoming event of the types the PE has subscribed to. It contains the processing logic of a PE such as updating a state variable. The API also provides a package for processing events as samples with time or count-based windows. This was unfortunately not ported to version 0.6, which was used in this work.

As with MapReduce, programming logic in PEs is conceptualized for one key as opposed to Storm where a Bolt can assume processing of the whole stream which may be partitioned based on the developers’ definition of the bolt in a topology. To include external streams the developer implements the abstract class `AdapterApp`. As opposed to Storm where Spouts which consume external streams are part of the topology, adapter apps in S4 are decoupled from the application logic so that they can benefit from properties of S4 apps such as scalability provided that they are applications themselves. Furthermore, it allows independent deployment and administration [Mor12].

S4 does not offer an abstraction layer as Storm does with Trident but the API offers packages for implementing time-based and count-based sliding windows which as previously stated where not ported to the current version.

4.3.4 Parallelism & Scalability

S4 uses MapReduce-like key-based partitioning of streams to achieve parallelism. Thus the degree of parallelism of each PE depends on the current number of unique keys of the consumed stream. Partitioning and distribution is automatically done by S4. The deployment is symmetrical, which means that all PNs contain all PE prototypes on all nodes of the defined cluster but PNs will hold different keyed PE instances. E.g. if we have two partitions and a set of 100 unique keys each PN would have 50 PN instances. To distribute the workload of each processing node one would choose to define the cluster with more partitions. Scaling is theoretically unlimited because there is no centralized node that can become a bottleneck.

If developers want to influence the number of instances of a PE, e.g. to a fix number of PEs, they can create a key set which is independent from the event data. For example, in our LRB application in S4 we first compute the segment statistics and emit the latest average speed and number of vehicles. This stream of events is then joined with the stream of events containing accident occurrences and the stream of position reports in the PE where the toll is calculated accordingly. In the `TollNotificationPE` we are able to use the vehicle id (vid) as a key, so that we have an instance for every vehicle. Keying on vid would not be feasible because we are joining all accident and segment statistics events on all segment a vehicle is driving. Hence every `TollNotificationPE` instance requires all the events from the accident and segment statistics PEs at least for the expressway for which it is calculating toll.

Therefore, we are left with two options. Either we can use the expressway (xway) and direction (dir) as a key or we create an artificial key. In the latter approach we divide the vid modulo n, the number of `TollNotificationPE` instances we would like to use for one expressway, and in the next step combine the result with the expressway and direction values to create a single new key. Hence, instead of using only two PE instances per expressway for emitting toll notifications
as suggested in the first approach, we partition the workload to TollNotificationPE, which is now keyed on a number between \([1, n]\) and the expressway and direction, which results in \(L \times 2 \times N\) number of instances. Figure 4.7 depicts the data flow for both approaches respectively when running the application for two expressways \((L=2)\) and an exemplary chosen \(N = 2\).

### 4.3.5 Fault Tolerance & Reliability

Fault tolerance and reliability in S4 is mostly achieved by the use of Zookeeper and is similar to Storm’s mechanisms. To assure high availability, more nodes than partitions need to be deployed on a S4 cluster. For example when deploying a cluster with 3 partitions and assigning 5 nodes to that cluster, one would have 2 standby nodes available, just like multiple worker slots are kept available in a Storm cluster. Zookeeper keeps a pool of all nodes which it has not assigned tasks to. As in Storm every node sends heartbeats to Zookeeper to communicate that it is still alive. If a node fails to send heartbeats and the node is not reachable it is considered dead, in which case a stand-by node tries to acquire the failed task to replace the failed node. Messages for the failed partition are redirected to the stand-by node, which now replaces the lost partition of the failed node. Since emitters specify only logical nodes when sending messages and are unaware of physical nodes only the mapping from logical to physical nodes has to be updated and events are rerouted to the new partition. This is similar to Storm’s fail-over mechanism.

For preventing loss of state accumulated over a longer period of time, S4 offers a checkpointing framework to periodically checkpoint the state of a PE. To minimize latency, checkpointing is executed in an uncoordinated way. This implies that meaning each checkpoint is taken independently without regard to global consistency and asynchronously. Asynchronous means that S4 tries to execute checkpointing without disturbing event processing. The checkpoint mechanism, as depicted in Figure 4.8 includes the serialization of the PE instance and saving it to a pluggable storage backend. Recovery is lazy, which means that it is executed only when necessary. This is
triggered when an event arrives at the replacement node whose key value is new. First the PE instance is generated, after which the last checkpoint for that key is requested. If successful the object is deserialized and copied to the new PE. The framework only checkpoints PEs whose state is marked *dirty*. Each PE instance is associated with a dirty flag which indicates whether or not the corresponding PE instance has been modified. This flag can be cleared and set at application level to e.g. avoid unnecessary checkpointing.

Since communication channels switched from using UDP to TCP reliability for message passing was increased but is not guaranteed. It is possible to prevent loss of events by using blocking senders and receivers for emitting events but the rate at which events arrive must be lower than the processing rate otherwise events are dropped. So opposed to Storm where message processing is guaranteed, a loss-free processing in S4 can only be achieved by a well balanced application design.
4.4 Summary

The systems introduced in the previous sections are used to implement parallel processing of large data streams. They offer a distributed execution environment for processing streams and are designed fault tolerant.

Both systems use similar concepts but with different terminology such as topologies and apps, bolts and PEs, adapters and spouts, tuples and events or partitioning and grouping. Even though the systems share a common purpose they differ at many points which is already reflected in the design approach. For example whereas S4 accepts loss of streamed data, Storm assures an at-least once and with the use of Trident API even an exactly-once-semantic.

Storm is younger than S4 and has a much more active, still growing community. This could be attributed to the fact, that Storm is well documented and offers better deployment and monitoring tools than S4. Submitting a topology is much easier than deploying an app in S4. Killing, deactivating and restarting is possible through one line command and the web-based UI. Starting an app in S4 requires multiple steps, including deploying a S4 cluster with the number of partitions needed, starting nodes on the desired machines, deploying the application and adapter before finally starting the adapter. Stopping and resubmitting an application is not possible and requires previous steps to be repeated. In Storm once a Storm cluster is setup, every other configuration e.g. the number of workers a topology needs to use is submitted with the application. The allocation and distribution of workers is done by Storm. The application can be monitored through the web-based UI which is helpful as well as needed to observe, for example which operators exceed its capacities.

S4 has a decentralized symmetric architecture. Tasks are evenly distributed among processing nodes. How many tasks exist in an running application is determined by the number of PEs and their associated keys. If only one value for a key, there will only be one PE instance. In Storm on the other hand the user has to specify how many times a spout or bolt is replicated. Tasks are then spread evenly among available nodes. An advantage lies in Storm’s ability to increase the degree of parallelism for bolt in a topology on runtime. Note that when rebalancing a topology leads to restarting the topology. Hence state accumulated in a bolt will be lost.

Both systems are modular and easily expendable and can scale high. The systems use key-based stream partitioning inspired by the MapReduce programming model to split a stream into multiple streams for parallel processing. However in S4 the degree of parallelism is only determined by the number of distinct values of a key and distribution is automatically done by the framework. Storm on the other hand allows different partitioning strategies by offering multiple grouping operators for stream partitioning such as shuffle-based partitioning or even direct allocation of the consuming task. Different parts of a Storm topology can be scaled individually by setting their own parallelism hint. This on the one hand makes developing applications in Storm very flexible and allows fine granular parallelization, on the other hand also adds more complexity and responsibility to the developer. Developing applications in S4 is simpler because of its more restricted programming model and the automated distribution of tasks.

Zookeeper is used in both systems for coordination and in their fault tolerance mechanism. An advantage is the optional but integrated checkpoint mechanism of S4, which allows partial state recovery in S4. A clear disadvantage is S4’s message delivery strategy: if the rate of incoming events is to high load shedding is applied. This restricts S4 from being used in certain applications which require stricter semantics. Storm on the other hand lets users define what tuples can be
dropped and which require guaranteed processing. If an anchored tuple could not be processed in a given time span it will be replayed from the beginning. The mechanism for tracking all tuples is designed to utilize memory as little as possible. This does not prevent loss of state though. If a task of bolt fails and is restarted, state, accumulated of previously successfully processed tuples is lost. To prevent this the developer has to implement her own checkpointing strategy. This again adds more responsibility to the developer. Another possibility is the usage of the Trident API where stateful processing with exactly-once semantics is possible. Trident as an abstraction layer comes near to offering a stream-based query language. Also the ability to submit ad-hoc queries with the use of DRPC is an advantage for Storm.

The following table 4.4 lists main aspects of both systems.

<table>
<thead>
<tr>
<th>Property</th>
<th>Storm</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Model</td>
<td>Data Stream, tuple-based, pull based</td>
<td>Data Streams, event-based, push-based</td>
</tr>
<tr>
<td>Programming Model</td>
<td>DAG of Spouts/Bolts</td>
<td>Actor Model, DAG of PEs</td>
</tr>
<tr>
<td>Architecture</td>
<td>distributed, master/slave</td>
<td>distributed, decentralized</td>
</tr>
<tr>
<td>Query language</td>
<td>Trident, DRPC, adhoc + continuous queries</td>
<td>none</td>
</tr>
<tr>
<td>Storage Layer</td>
<td>data partitioned based on grouping by user</td>
<td>data partitioned based on key/value pairs</td>
</tr>
<tr>
<td>Parallelisation</td>
<td>task parallelism set by user</td>
<td>automatic</td>
</tr>
<tr>
<td>Scalability</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Reliability</td>
<td>Rollback Recovery, (Upstream backup)</td>
<td>Checkpointing</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Failover mechanism</td>
<td>Failover mechanism</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of Storm and S4.

Nevertheless both systems are only frameworks. While the majority of the aforementioned systems offer query languages with built in stream-based operators, S4 and Storm offer only skeletons for user defined functions. On the other hand operation on data are constrained by the set of operators available in said languages. None of the system have a storage layer integrated and scaling is mostly determined by the users configuration. Thus not all requirements for real-time processing introduced in section 2.4 cannot be met. However their pluggable architecture allows easy integration of with other components such as databases. The user can create a customized architecture herself, which can be seen as a curse and a blessing.
Chapter 5

Experimental Evaluation

In this chapter the results of running the Storm implementation of LRB are presented. First the main goals of the experiments conducted will be explained, followed by a description of the adapted LRB and its associated datasets. Before presenting the results of the experiments, each implementation will be briefly illustrated and the environment and tools used specified. In the discussion section I will review the problems I encountered while implementing the LRB, what limitations the implementations and experiments have and how they could be overcome.

The experiments which are dealt with in this chapter aim at showing the improvement in performance of the previously introduced systems compared to other distributed systems. In the process it will become clear whether recent systems are more suitable for processing large-scale data streams than non-parallel systems. The main goals of this evaluation are to investigate:

- How effective is exploiting inter-operator parallelism in an SPS?
- By what factor can a parallel SPS outperform a distributed DSMS?
- Which system has a better performance?
- Can both systems meet the requirements for a real-time SPS introduced in section 2.4?

5.1 Methodology

Originally the LRB as described in Section 2.5 was implemented to identify the factor by which a DSMS can outperform a DBMS. The main goal of this evaluation is to support the statement that a parallel SPS can outperform a distributed DSMS. As a reference I chose the LRB implementation of SPC [Jai+06], mentioned in Section 3.3. Their implementation of the LRB consists of 17 PEs. In their experimental evaluation they achieve an L-rating of 2.5 when running on a single node, which corresponds to the results of the first LRB implementation in Aurora. They do not present an L-rating of their distributed experiments but describe different configurations with different numbers of expressways. When conducting the benchmark with a PE-per-node approach, for example, they achieve an L-rating of 5 with 17 nodes utilized.

The LRB homepage [Bra] provides a program that generates the input for the benchmark, which is traffic simulated over a period of three hours. The input is saved in a flat file and contains tuples marked with timestamps reflecting the times of their generation. There are three⁹ types of events:

⁹Travel time estimation requests are omitted.
5.1. METHODOLOGY

- **Position reports** are tuples in the form of (Type=0, time, VehicleID, Speed, XWay, Lane, Direction, Segment, Position).

- **Account Balance** requests are in the form: (Type = 2, Time, VehicleID, QueryID)

- **Daily Expenditure** requests are tuples in the form of (Type = 3, Time, VehicleID, XWay, QueryID, Day)

The input data for a single expressway L=1 consists of around 12 million position reports, 60000 account balances and 12000 daily expenditure query requests. During the simulation the input data is increased as can be seen in Figure 5.1.

The **datadriver** provided by the developers of the LRB, which is used to deliver input data in a manner consistent with the tuple’s timestamps, gave rise to a number of problems and could not be used. A new data driver had to be developed. As evaluation was I decided to implement a TCP-socket-based server java program with the same functionality as specified by the LRB developers.

Since problems with the validation tool provided, which is used to verify the output of the simulation, could not be resolved, I decided to make an adjustment to the requirements to simplify verification of the output. This includes that every position report triggers output of a toll notification so that the number of toll notification is equal to the number of position reports. For this reason this work will not present an official L-rating but instead use L as a workload metric.

The systems are evaluated by running each LRB application with different configurations in deployment. To determine if the systems achieve linear **scalability** first I increased L by one. When the requirements could not be met, I increased the number of workers in Storm and partitions in S4 until the requirements of the LRB were met again. This was repeated until all available nodes are in use.

The Storm experiments were conducted on a cluster of 4 nodes, each using two Xeon E5-2620 2GHz

---

\[10\] Verification of toll notifications failed even though toll notifications which the validator listed as missing were contained.
processors with 6 cores. Each has 32GB of main memory, two 1TB SATA disks and is connected to a 1GBit Ethernet switch. All nodes have Ubuntu Server 12.04 LTE installed and use version 1.7.0 of the Java Runtime Engine. Java heap space was initially set to a minimum of 512MB and a maximum of 3GB, but when testing the application sudden high peaks in processing latencies occurred towards the end of the simulation, which later normalized again. This can be attributed to the fact that a lot of CPU time was spent on resizing Java heap space. To avoid performance loss for increasing Java heap space, initial and maximum heap sizes were set to 2GB. Experiments for S4 were conducted on MacBook Pro Dual-Core Intel Xeon 2.66 GHz, with 4 GB of main memory and one 500GB SATA disk.

5.2 Limitations

Originally a detailed performance comparison between Storm and S4 was planned. Due to a restriction of internet access, however, it was not possible to build S4 apps on the test cluster. Building the app locally and deploying it on the remote cluster was not successful. Since the issue could not be resolved, experiments regarding S4 were conducted locally and are restricted to a single machine. To allow for a comparison of both Systems, the implementation of Storm was run on the same machine. However the performance requirements of Storm are higher than in S4 and tests could not be run sufficiently on the same machine. Therefore a detailed comparison was not conducted.

In the course of running the experiments, it turned out that the performance of the above mentioned datadriven was insufficient. As of a workload with more than 4 expressways, delivery of tuples was too slow. Hence the scale-factor of the experiments is limited to 4.

5.3 LRB Implementation

StormLRB. Figure 5.2 shows the data flow graph of the implementation. The implementation consists of the following operators:

1. StreamBolt consumes the input stream.
2. SplitBolt splits the stream into three different streams each for every tuple type.
3. SegmentStatisticsBolt calculates the average speed in a segment over a window of one minute.
4. AccidentDetectionBolt detects accidents.
5. AccidentNotificationBolt alerts vehicles entering the proximity of an accident area.
7. DailyExpenditureBolt consumes and responds to daily expenditure queries.
8. FilePrinterBolt collects notifications, persists them to file. There is one FilePrinter for each request (toll, accident notifications, account balance and expenditure answers).

Because none of the systems in question have a storage layer integrated, I used a stub which simulated answering to daily expenditure requests. This is acceptable, since first test runs while
implementing the application showed, that the bottleneck operators of the application are computing the segment statistics and toll notifications as well as the accident detection as well as writing to file.

In order to visualize what effect exploiting data parallelism has on performance, two configurations of the Storm topology are used. As described in Section 4.2, Storm has an open programming model and developers have multiple possibilities to parallelize workload. To find an optimized configuration, first I ran the implementation with a parallelism hint of 1 for each bolt. Then I identified the bottleneck operators and parallelized execution of the bottleneck bolts with different parallelism hints and initial task numbers until I observed an overall balanced execution. In an optimized approach I used partitioning on two levels, using the optimum configuration of the first approach, which was then replicated for each expressway, so that every expressway created a new sub-graph. Figure shows an excerpt of each approach.

Figure 5.2: StormLRB Linear Road Benchmark Implementation.

Figure 5.3: StormLRB Linear Road Benchmark Implementation.

Figure 5.3: StormLRB application with two partitioning approaches.
S4rb. S4’s implementation depicts an adapted version of the LRB. S4rb computes the two continuous queries toll notification and accident alerts. It consists of the following operators as depicted in Figure 5.5.

1. InputAdapter consumes the input stream.
2. DispatcherPE a keyless PE consumes above stream, creates position reports events and dispatches the stream to multiple downstream PE.
3. VehicleSpeedPE is keyed on expressway, segment and direction and mod 100 and calculates average speeds of vehicles.
4. SegStatisticsPE is keyed on expressway segment and direction and calculates the average speed in a segment over a window of one minute.
5. AccDetectionPE is keyed on expressway and direction and detects accidents.
6. AccNotificationPE is keyed on expressway and direction and alerts vehicles entering the proximity of an accident area.
7. TollNotificationPE is keyed on expressway, direction and vid%100 and calculates toll and notifies vehicles of that toll.
8. FilePrinterBolt is a singleton instance and collects notifications, persists them to file. There is one FilePrinter for each request (toll and accident notifications).

5.4 Results

The performance of the StromLRB application was first ran with a parallelism hint of 1 for every bolt. Storm achieves a workload of 2 when one worker slot was used. This is similar to other none parallel implementations of the benchmark, where a maximum of L=2.5 was scored when utilizing one core. The average latencies for processing a position report until emitting toll notification lies with a maximum of 726ms well within the requirements of the LRB. Using the the same

---

11I did not have data for L=2.5 available but I assume that the application can achieve an 2.5 as well.
configuration, to process 3 expressways could not satisfy the latency requirements. Storm is able to process the workload of 4 expressways on a single machine utilizing 2 worker slots. Figure 5.5 shows the average latencies for every second of the simulation when running StormLRB and S4Lrb with L=4 and two workers and node, respectively.

Looking at the standard deviation one can observe, that S4 latencies are more similar, which means that the system is able to balance load consistently. Storm's processing latencies on the other hand vary significantly. This can partly be attributed to the use of the Storm UI during the experiments. The performance obviously increases as the total number of cores increases. However, the tests were not sufficient enough to establish the scale factor for any system. Both partitioning approaches did not show differences. The seemingly optimized one appears to have even more workload than the non optimized one. Possibly the implementation was not done properly since the amount of processed tuples as indicated in the toll notification bolt doubled or tripled respectively, depending on how many express ways have been used.

Eventhough no complete comparability is ensured, because the benchmark is not fully implemented and was run on a more performant system, one still can conclude due to the high performance and low latencies (compared to e.g. SPC with up to 1.79s of that the systems both outperform a distributed SPS implementation.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Min. (ms)</th>
<th>Mean (ms)</th>
<th>Max. (ms)</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>StormLRB L2</td>
<td>0ms</td>
<td>29.86</td>
<td>1690.00ms</td>
<td>21.78</td>
</tr>
<tr>
<td>StormLRB L4</td>
<td>0.3713</td>
<td>9.3556</td>
<td>539.0319</td>
<td>14.130</td>
</tr>
<tr>
<td>S4Lrb L4</td>
<td>2.308</td>
<td>25.682</td>
<td>799.808</td>
<td>7.6821</td>
</tr>
</tbody>
</table>

Table 5.1: Examplary values for latencies when running the benchmark in S4 and Storm.

![Figure 5.5: Latencies of StormLRB for L=2.](image)

S4 could fulfil the latency target but did not match the remaining benchmark requirements. The number of toll notifications was not equal to the number of position reports. Hence S4 could not successfully perform the benchmark.
Chapter 6

Conclusion

Processing large scale data streams has become prevalent in many areas. These vast amounts of data can only be processed by distributing the workload across many machines. This work focuses on analyzing requirements and approaches for distributed stream processing. In Chapter 2 I outlined basic terms related to large scale data processing, most importantly parallel and distributed computing. I introduced the complementary paradigms of batch and stream processing and established the requirements for real-time processing systems. The Linear Road Benchmark was introduced and used as an example throughout this work to deepen the understanding of the concepts which were introduced and which relate to large scale data processing in the following chapter. While MapReduce allows to analyze large batches of stored data, the data stream model offers the opportunity to continuously process unbounded streams of data online. The processing of large scale data streams, however, requires the distribution of work-load. The systematic challenges of distributed stream processing have been briefly highlighted. As a result, three distinct but not mutually exclusive approaches of distributed stream processing:

- micro-batched stream processing,
- inter-operator parallel stream processing
- data parallel stream processing

Two systems, Apache Storm and Apache S4, were chosen to represent the third approach and were made subject to detailed evaluation. These frameworks allow for “easy” parallel programming, which can be used to create scalable and fault tolerant applications. In the process I focused on determining what fault tolerance mechanisms they have and how parallelization can be accomplished. Both Storm and S4 rely on key-based partitioning of data streams inspired by MapReduce but they differ in their programming model. Storm offers an open programming model but lets the developer handle such issues as load balancing and setting the degree of parallelization. S4 on the other hand has a strict partitioning strategy and automatic load balancing. The Storm Trident Api is an example that above approaches are not mutually exclusive.

In Chapter 5 I evaluated the different performance and scalability abilities of the two systems by implementing the Linear Road Benchmark. This was only partially successful as we were not able to create sufficient system load to see the system actually scale. Still, the benchmark shows a clear tendency for both systems. They both can process data streams faster than systems utilizing only inter-operator parallelism. By partitioning query operators the systems achieve high throughput rates while computing complex queries. Furthermore, we have assessed that the requirements for real-time processing systems, which were previously specified, are only partially met by the systems. This is mainly due to the fact that they are frameworks such as MapReduce which focus
on providing skeletons for parallel continuous processing of data streams rather than complete systems. It is one of the developer’s responsibilities to handle stream imperfections, out of order data, etc. This can also be beneficial, however, because the developer can implement those things only if necessary, which reduces overhead in processing.

S4 lacks the higher level support of a composition language, which is provided in Storm Trident. This makes Storm more accessible to a wider range of users who need stateful continuous processing. Both systems involve a modular design, however, and can easily be complemented, which can be an advantage compared to complete systems, which will offer limited options only.

Although a considerable amount of analysis has been done, additional questions still remain to be answered. How effective is the system’s utilization of resources in comparison with the other? Where are the limits of performance and scalability? How would Storm Trident, which offers the micro-batch processing approach on top of Storm compare? Furthermore it may be worth testing the potential impact of the systems fault tolerance mechanisms on processing.

Writing reliable scalable distributed applications is challenging even though the systems make this process easier. In addition, it has become apparent that there is the need for programming models to support multiple levels of coding, which need to be lower if more efficiency is needed but higher if problems are straightforward. The Linear Road Benchmark is a thorny issue and needs a lot of time to be customized to the required task. The time I spent getting the benchmark tools up and running and completing the implementation would have been needed for executing and analyzing results. Against this backdrop, I only accomplished a rather rough impression. Selecting a micro-benchmark may have been an option that could have allowed for a more detailed analysis. The scope of this thesis would be dramatically increased if it were to encompass a third performance evaluation such as SPark Streaming or Trident. In any future thesis it would be fruitful to entirely focus on how to design a feasible benchmarking for distributed systems.
Bibliography


[S4j] Apache S4 0.6.0-incubating: API Javadoc. URL: http://people.apache.org/~mmorel/apache-s4-0.6.0-incubating-doc/javadoc/ (visited on 04/01/2014).


Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, 24th April 2014