HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Master of Science in Engineering

Data Stream Management Systems: **Apache Spark Streaming**

Seminar: "Advanced Database Systems"

Hochschule für Technik Rapperswil

Herbstsemester 2016/2017
February 20, 2017

Author: Martin Stypinski

Supervisor: Prof. Stefan F. Keller

Abstract

This paper is the result of the 'Advanced Database Systems'seminar at the University of Applied Sciences in Rapperwil. The key point is to explain and understand the function of data stream management systems. The paper is split into two parts to cover a theoretical and a practical aporach by evaluating one particular product. The first part will cover all the basic concepts and reason the technology. The second will explain the concepts by implementing them in Apache Spark Streaming. A small benchmark will be provided to compare the results between other participants.

Contents

Abstract i

Inhaltsverzeichnis ii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 ACID vs CAP . . . 3
- 1.3 Data Stream Management System . . . 5

2 Apache Spark 7
- 2.1 Overview . . . 7
- 2.2 Apache Spark Streaming . . . 11
- 2.3 Eight requirements for Data Streams . . . 12
- 2.4 Java Example . . . 15

3 Implementation 16

4 Benchmark 18

Appendix 20

List of Figures 21

Glossary 22

Bibliographyx 23

1 Introduction

This paper is the result of the seminar on "Advanced Database Systems" at the University of Applied Sciences in Rapperwil. The paper covers all the necessary concepts of "Streaming Database Systems" and investigates one particular product for its abilities. The product is determined to be Apache Spark Streaming which is well known to be used in companies handling big data streams for daily business like Uber[2], Netflix and Pinterest.

1.1 Motivation

Modern Relational Database Management Systems (RDBMS) show the perfect snapshot of the past, while modern businesses heavily rely on data, with these requirement these concepts become limiting. While there is a need for storage and persistence, where RDBMSs are still in their prime, there is another need for real time data processing. Real time data processing can be useful to give an accurate insight in the moment and anticipate the future on shorter notice. This use case is a driving force to justify a new database concept for data streams - the Data Stream Management System DSMS.

Example As an introductionary example that will be used through the paper a simple online store and order process was taken. While beeing interessted in a few new cloths, a customer is adding 3 T-Shirts into his basket. The basket has now a value of 48 USD and having a value of 50 USD enables him to get free shipping. Therefore he exchanges a T-Shirt with another fancier one. The basket has now a value of 55 USD.

The RDBMS approach In modern online stores the most common domain model pattern of choice is the Transactional Line Item pattern 1.1. Each item in the basket is represented as an entity and the basket is represented as an entity as well which is mapped to a customer and few more states. [6] This representation is very accurate and proven to show the final order state.

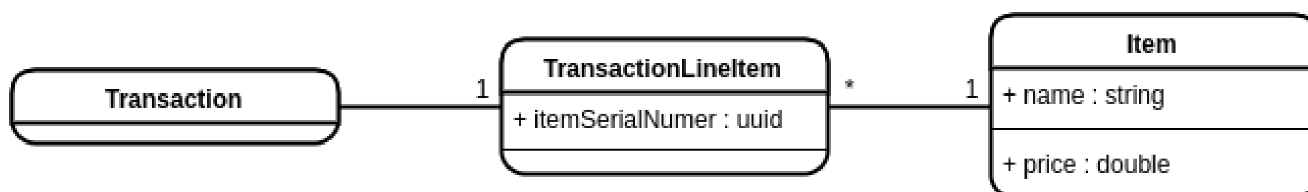

Figure 1.1: Transactional Line Item Pattern - to implement an order state

At this stage it is possible to collect the final order state and prove correlations within orders throughout the customer base. One of the most famous examples have shown that young men tend to buy daipers

and beer together on fridays. The interpretation is left open but it is assumable that they are watching football, while babysitting the kids without their wife beeing at home [5].
The example order at this point is just indicating the final value of 55 USD. But there are more insights that can be explored. The customer was struggling to fullfill the 50 USD goal to be eligable for free shipping. All these valuable customer actions can hardly be mapped in a RDBMS as the actions represent a constant change during the order process.

The DSMS approach Beneath the rigid entities stored in the RDBMS there is more information to be discovered. The small changes made to the basket, to reach the customers goal of 50 USD, are easier to represent in an event based way. These changes to the final order can be represented as a series of events as a so called data stream. 1.2 With a data stream our online store can be elevated to an event driven implementation. The event driven design is generating more data and therefore there are many more factors, dependencies and decisions to be analyzed. The final basket is put together by many small events that changed the baskets state. Every single event can be accesed on its own - but in the meantime the basket is represented in its final stage. Patterns in behaviour can therefor be discovered more easily.

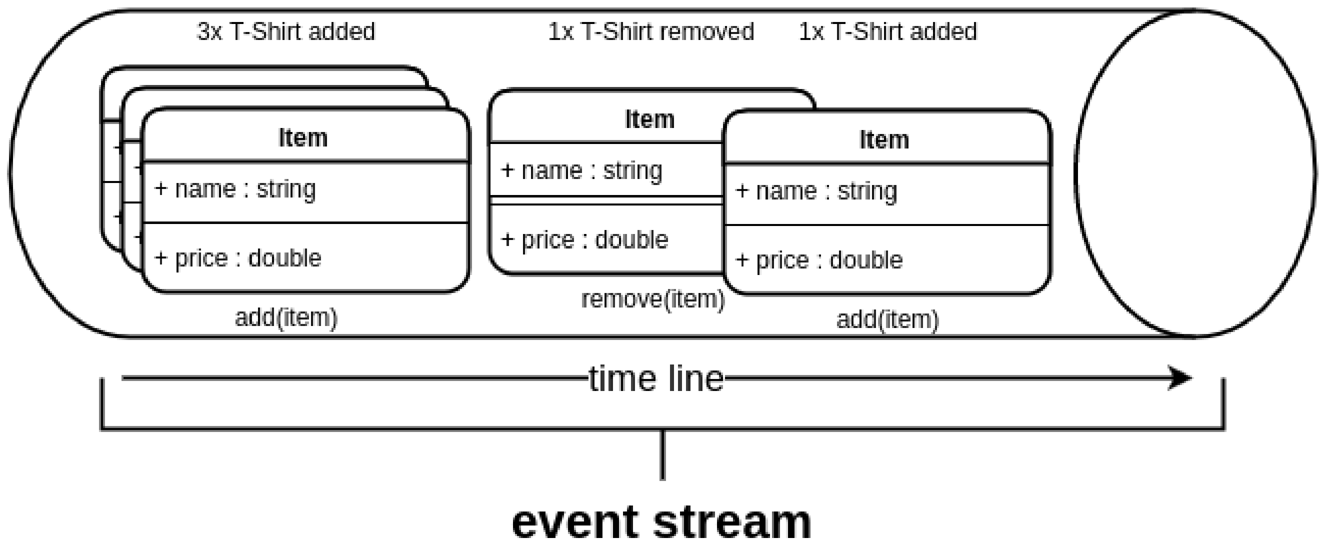

Figure 1.2: Event Stream - changes can be made visible

1.2 ACID vs CAP

To understand and justify the need for new concepts, it is necessary to have a look at already given and proven concepts and its limitations. Fundamentals of database systems require to understand the concept of the CAP-Theorem. The CAP-Theorem was first introduced by Eric Brewer in 2000 and later proven to be accurate for RDBMS.

> When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. (1.3) It is impossible to achieve all three. [4, S. 1]

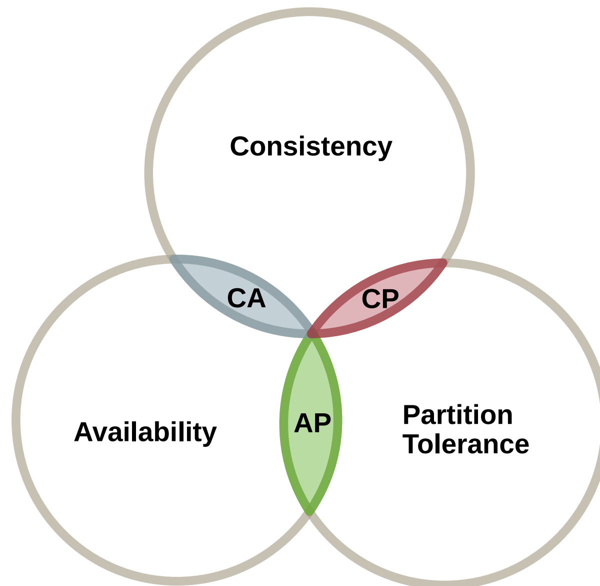

Figure 1.3: CAP-Theorem with its possible overlaps

- **Consistency:** Guarantees that a read will receive the most recent data or an error.
- **Availability:** Every request will get a response, although no guarantee on correctness of the data.
- **Partition Tolerance:** The system can operate although parts of it are unavailable due to network or hardware failure.

In fact every software engineer dreams of a system that incorporates all three attributes, but as it is almost impossible to provide - two out of three are the rule.

	Consitency	Availability	Partition Tolerance
Consistency			
Availability	**Relational** PostgreSQL, MSSQL, Oracle Database		
Partition Tolerance	Neo4j, Redis, Google Big Table, MongoDB	CouchDB, Cassandra, Amazon Dynamo	

According to this table, relational databases, are very well designed to store data and make them consistent-available. This leads to an architecture similar to the following illustration. 1.4

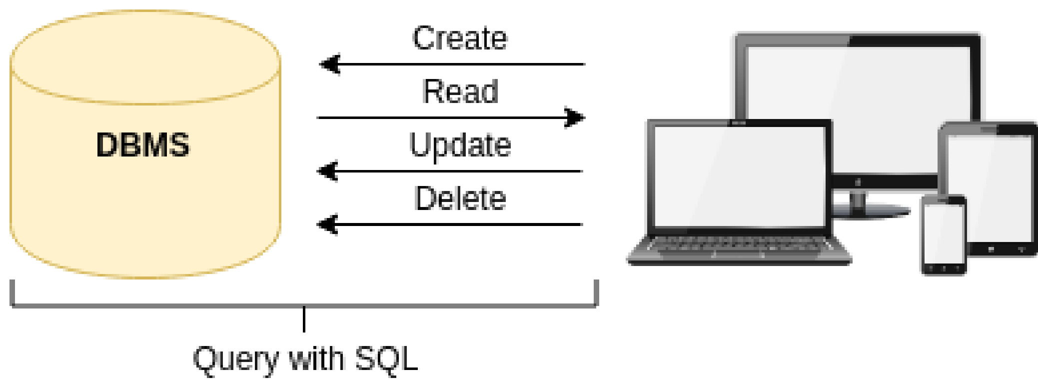

Figure 1.4: CRUD operations in a DBMS

Relational databases and the CRUD architecture are very capable of storing states mapped to entities, but they lack a little handling of continously changing values. The theorem is indicating limitations to a certain degree, with the requirements of processing streaming data it becomes necessary to have the data available and consistent, but at the mean time partition tolerance has to be given in some way to guarantee scalability. These fundamental requirements and a few more were predicted by the "The 8 Requirements of Real-Time Stream Processing" [7] paper.

1.3 Data Stream Management System

To enable our t-shirt store to gather more information about a customers behaviour, a new system has to be introduced. The implementation is event based and will be handled by a DSMS.
Despite scalability and realtime data processing there is another driving force for innovation in database systems. Technology itself is becoming more and more troublesome to store data at this kind of load. Computational power is getting more and more powerfull each day, memory prices are at an all time low and storage is not expensive - but storage is still not up to the task to handle that load. (1.1)

	1987	2004	Increase
CPU Performance	1 MIPS	2,000,000 MIPS	2'000'000 x
Memory Size	16 Kbytes	32 Gbytes	2'000'000 x
Memory Performance	100 usec	2 nsec	50'000 x
Disc Drive Capacity	20 Mbytes	300 Gbytes	15'000 x
Disc Drive Performance	60 msec	5.3 msec	11 x

Table 1.1: Historical development of storage - Source: Seagate Technology Paper

It is inevitable that the best way to store data is in big chunks to avoid the disk as a bottleneck. Incoming data should be handled in almost realtime and then be stored later to a disk, so important actions can be taken just in time and the data is at a later stage available for later evaluation.
These necessities point the architecture towards new concepts.

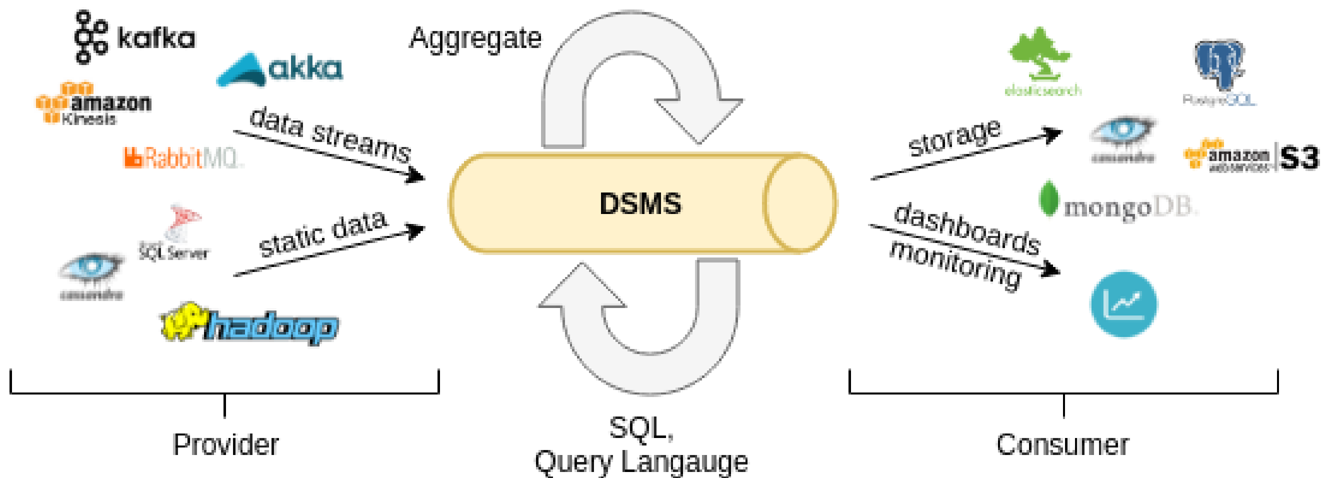

Figure 1.5: Schematic of a DSMS architecture

This change enables, in the mentioned online store example, to gather more data on customer behaviour and understand how a customer is actualy ordering the items individualy. The new architecture enables to aggregate data in almost real-time and is able to store it as aggregated entities in different database systems.
Many applications can greatly benefit from this architecture, a few examples:

- **Sensor Aggregation:** Sensor data can be aggregated and visualized on a dashboard, but the raw data is stored to evaluate the data for quality assurance purposes later
- **Authentication:** As realtime data is flowing, user behaviour can be analysed and unexpected access can be detected in real time. (Example: A accountant is responsible for the Singapore branche of a major bank. At a certain point he is accessing customer data in the Hong Kong branch. Why is he doing this?)
- **Pricing and Analytics:** As there is major blackout in an area, people start to buy candle - within secounds the prices can be adjusted because the instant demand for candles can be monitored.

DSMS are usefull where real-time, event based, continious querries become a central question and Apache Spark Streaming is just product that can handle it well.

2 Apache Spark

2.1 Overview

Although Apache Spark is an open source framework for cluster computing, it can be used as a DSMS as well. The framework was developed at the University of Berkley and in the the year of 2013 it entered the Apache incubator. At first it was a proof of concept for the resillent distributed dataset (RDD) and through time it grew to the modular product it is nowerdays. [3]

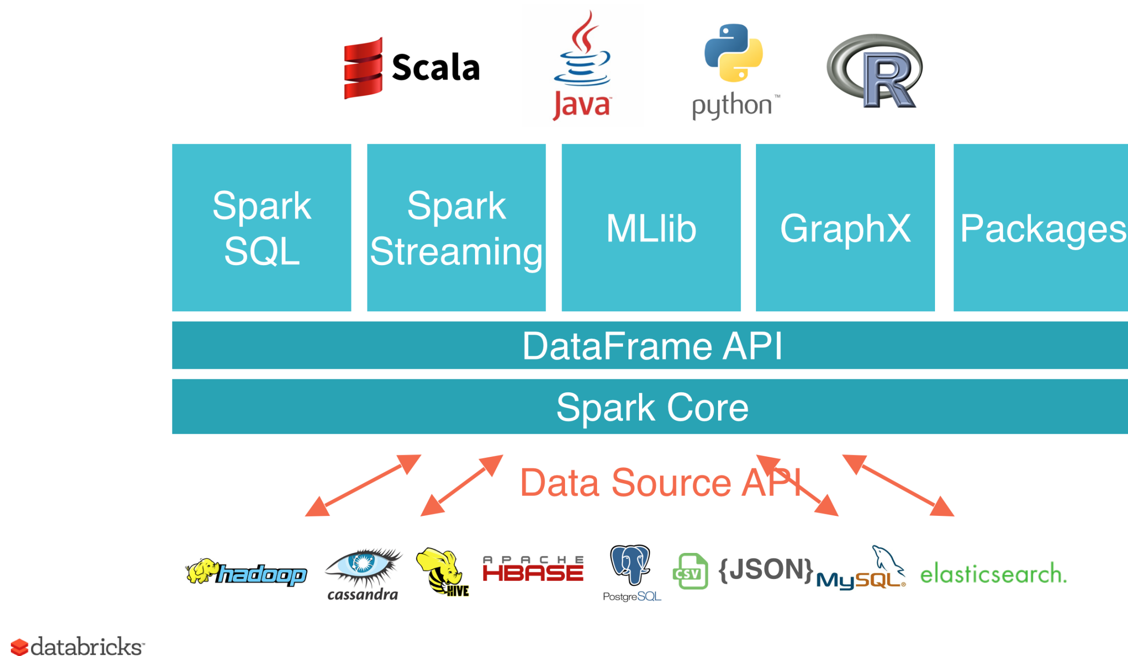

Figure 2.1: Apache Sparks and its plugins - Source: Databricks

Apache Spark is not build to made to communicate with Apache Kafka or used for data streams, but through its modular architecture (2.1) it has the flexibility to solve these challenges easyily. To understand the reasons for implementing RDD and its benefits it is important to understand what is happening under the hood of Apache Spark.

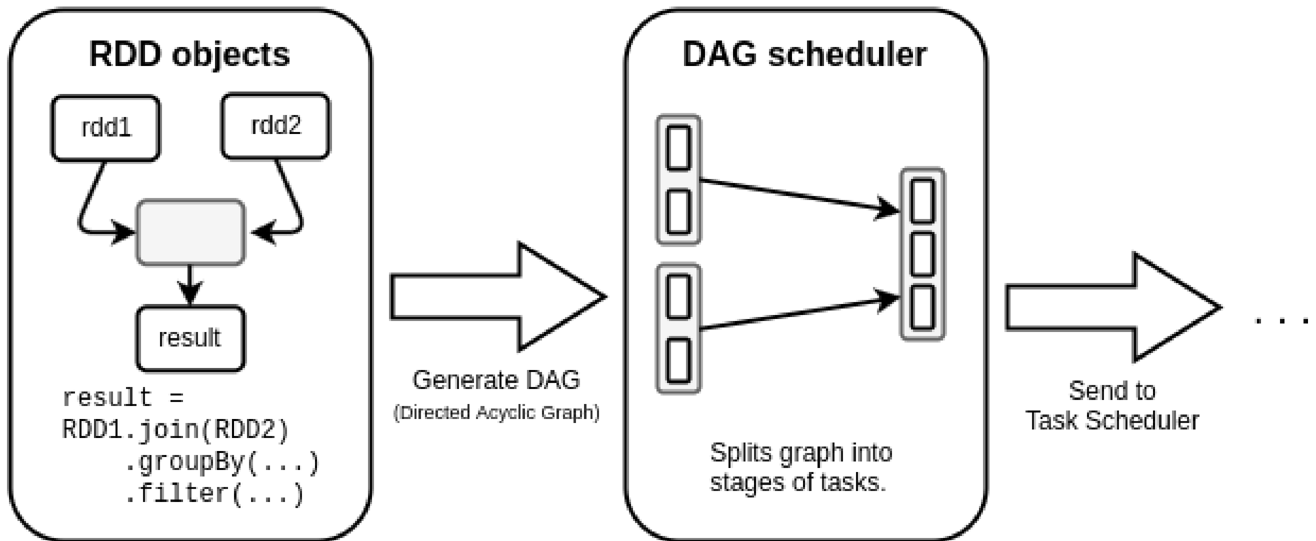

Figure 2.2: Apache Sparks internal architecture

There are numerous components needed to interprete and run the code sent to Apache Spark. For deeper perception it is beneficial to understand the first two layers.
Given a simple Java example (2.2) the first layer interpretes the provided code. Spark uses internally a Scala interpreter with some modifcations and is therefore compatible with all JVM languages. As the code is parsed, the needed amaount of resillent datasets are generated. At this point it should be mentioned, that the RDDs are carrying objects for data and operations.
The second layer disects the RDD structure into a directed acyclic graph (DAG) which can be distributed over multiple cores or clusters. The DAG is very important to determine parallelism and the sequential order! This highlevel approach enables an interpretation of the actual graph, which makes cashing and optimizing on a high level feasible.
The next steps involve scheduling and distributing, the distribution is done as an actor model within the Akka framework. Understanding the detailed process does not add any value to the general understanding of Spark and therefor a detailed explanation is left out. The neatness of this solution is hidden in the RDD implementation and its benefits over a map reduce pattern.

Map Reduce The map reduce pattern (2.3) is very common for writing highly parallelizable applications and often found in big data problems. The concept is very simple but efficent, by distributing the map-operation over several cores or even clusters and reducing the data structure at a later point. This makes an expensive operation highly parallizable.

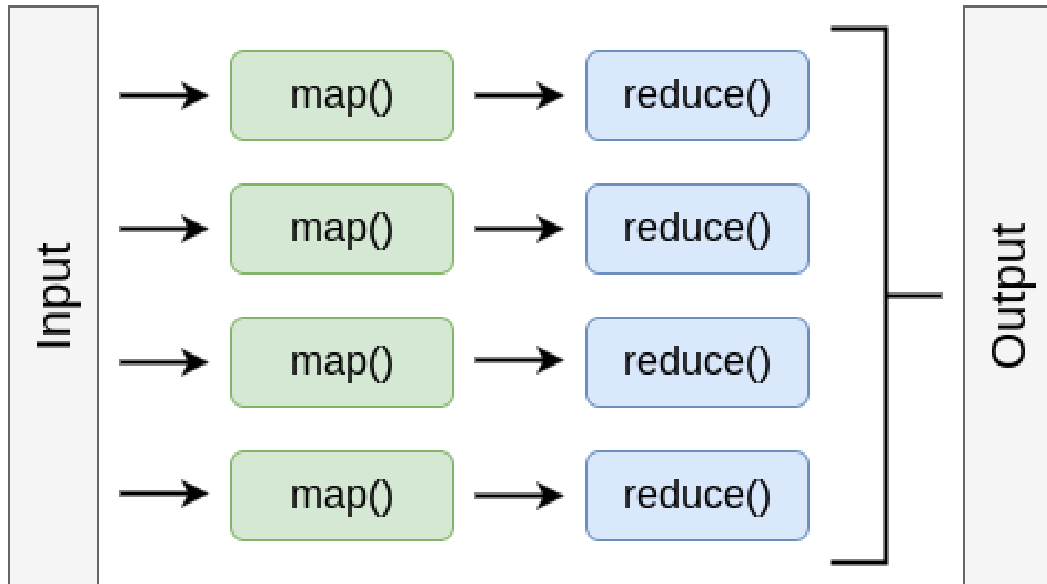

Figure 2.3: Schematic of the map reduce pattern

RDD - Resilent Distributed Data The biggest disadvantage of the map reduced pattern is its strict structure. Every dataset has to do the same operation. This problem was adressed with the resilent distributed data structure. Instead of applying the same operation to each data set, the RDD structure encapsulates chunks of data and an so called 'Action' or 'Transformation' into a lightweight datastructure before distributing it.
In contrast to map reduce RDD can perform actions and transaction similar to 'Java Collections'. (2.4) It gives the developer more freedom and flexibility in implementing algorithms. Further RDDs are immutable and therefor cachable. At each point in time there is a new RDD generated and if necessary cached for the the next operation. This gives a solid boost in performance on certain operations.
Subsequently, RDD is less rigid compared to map and reduce and closer to simple development concepts through its 'Java Collections' concept. It simply wraps data with a set of operations to a highly distributable object. RDD offers one particular disadvantage which is neglectable nowerdays, due to hardware pricing. The objects have to fit in memory in order to keep the efficency. Due to its recipe behavior it is not possible to store the data on a file system in between the operations.

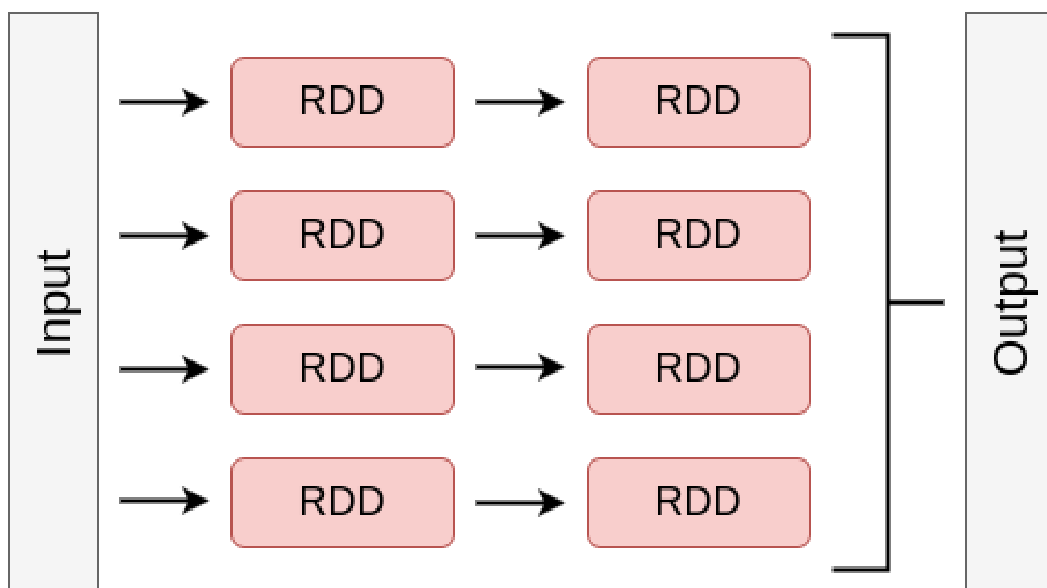

Figure 2.4: Schematic of RDD based processing

The motivation for Spark as a distributed computing framework is given. These RDDs are the building blocks for Apache Spark and are the reason why the data stream needs to be discretized through Apache Spark Streaming in order to be processed.

2.2 Apache Spark Streaming

Apache Spark was initialy not developed for its streaming data capability, but more as a proof of concept to implement RDD. Over the time, many extensions have made this framework to a very diverse streaming data framework, machine learning framework and graph processor.

Discretization Apache Spark can handle actions and data in a very performant way. But to enable it to handle realtime data at all, it needs to be split into batches. (2.5) Due to this architecture Apache Spark Streaming is responsible for discretizing the input stream into a so called DStream (Discrete Stream). A DStream is an abstract concept this process. It is a Java Class at the same time, which gives access to this discretized data.

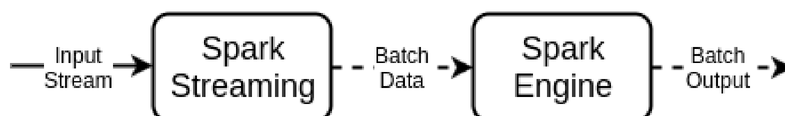

Figure 2.5: Spark Streaming feeding batch data to Spark Engine

Once the stream is discretized by Apache Spark Streaming it is feed into the Spark engine in small batches (micro batches). These micro batches are processed and the output is generated in batches as well. The discretization is a necessary evil to make Apache Spark perform with constant flowing data.

Computing The abstracted DStream is representing the incoming stream in batches. (2.6) These batches of data are handled as RDDs in Apache Spark. For each time slice there are RDDs with the according data.

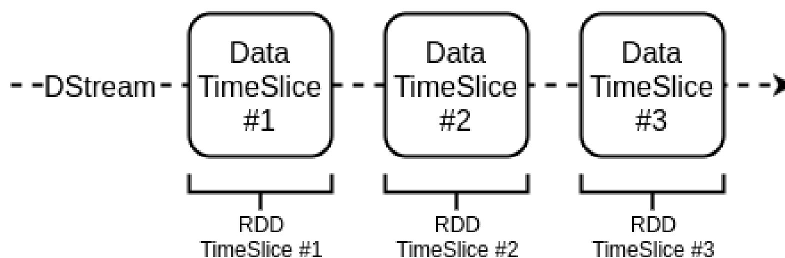

Figure 2.6: DStream representation of multiple RDD

These measurments are neccessary to make Apache Spark to a performing DSMS and fullfill the reqirements once written down by Michael Stonebraker et al in a paper. [7]

2.3 Eight requirements for Data Streams

Long time before the technology was available, Michael Stonebaker et al. formulated 8 requirements to handle streaming data efficently. [7] The paper claims the nececity for a data stream based database technologie already in 2005. Apache Spark Streaming was released 2014 as a potential implementation. The requirements show a baseline for DSMS and are a good indicator how well Apache Spark covers holds up as a DSMS.

Rule 1: Keep the Data Moving

> *The first requirement for a real-time stream processing system is to process messages "in-stream", without any requirement to store them to perform any operation or sequence of operations. Ideally the system should also use an active (i.e., non-polling) processing model.*

The requirement to handle datastreams in a realtime maner, indicates that there is no prior storage to the processing of the data. Apache Spark implements a so called DStream which is represented by an immutable data stracture. The immutability of the stream guarantees that reads are always consistent and for this reason a read has never to wait for a write on the stream. There is no such thing as a update operation, there will simply be a new object and the stream can not be blockt. Underneath, the implementation is based on microbatches - time sliding window based snapshot system to determin the current data set. [1]

Rule 2: Query using SQL on Streams (StreamSQL)

> *The second requirement is to support a high-level 'StreamSQL' language with built-in extensible streamoriented primitives and operators.*

Data on the stream should be handeled in a high-level language. Apache Spark Streaming is an extension for the Apache Spark Core, which implements numerous high-level APIS for Python, Scala and Java. Further more there operations can be exectued in SparkSQL or HiveQL. [1]

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

> *The third requirement is to have built-in mechanisms to provide resiliency against stream 'imperfections', including missing and out-of-order data, which are commonly present in real-world data streams.*

To handle stream imperfection a so called 'event-time' has to be defined. Network topologies, bottlenecks and other factors can cause data delay. The event-time as a global time stamp guarantees the dataframes order in the stream.
Since Apache Spark 2.1.0 there are several advanced methods to handle streaming imperfections such as the late data threshold. This threshold gives late incomming data still the ability to be counted in the right time window. [1]

Rule 4: Generate Predictable Outcomes

> *The fourth requirement is that a stream processing engine must guarantee predictable and repeatable outcomes.*

Quite contrary to different streaming database systems, Apache Spark uses microbatches of data within each time slice. These microbatches contain the actions and the data needed to fullfill the steps. The actions in the RDD are deterministic and therefor equality of data implies equality of results. Determinism is given!

Rule 5: Integrate Stored and Streaming Data

> *The fifth requirement is to have the capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data.*

Apache Spark offers various storage engines which can be used to provide data. Apache Spark Streaming at the mean time can be used to handle an incomming stream and these chunks of data can interact with each other.
A typical application for this usecase is fraud detection. Legitimacy of user interactions are evaluated based on realtime data in comparison to already known patterns in the data store.

Rule 6: Guarantee Data Safety and Availability

> *The sixth requirement is to ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures.*

Fault tolerance is guaranteed through reciever logging, worker logging and checkpoint logging. (2.7) The Executor is responsible for recieving the data and forward it as block data in memory. This block data can easily be distributed.

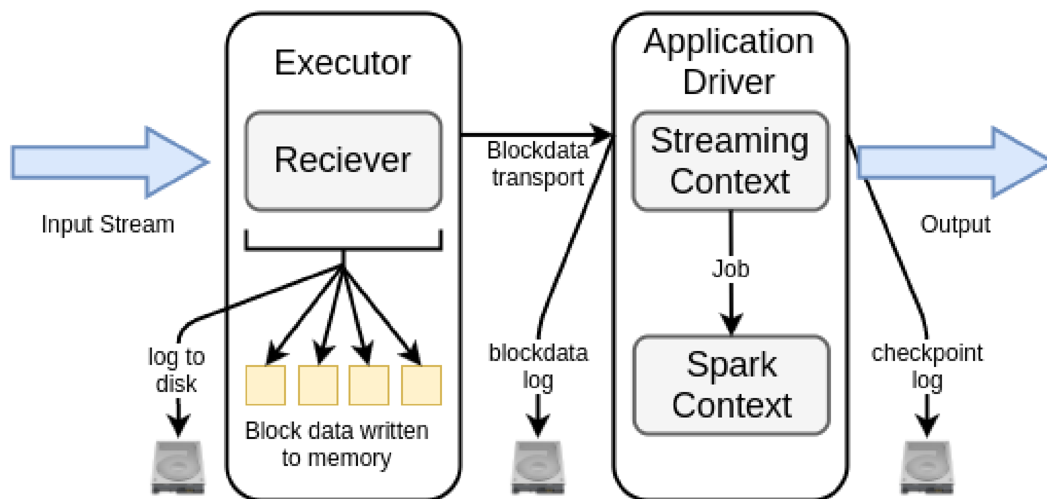

Figure 2.7: Fault tolerant logging for fast recovery

- **Reciever Logging:** Data is written to a disk similar to a write-a-head log in modern RDBMS systems.
- **Worker Logging:** The worker recieves the data and stores the block data to the disk. This gives an advantage in the case of recovery
- **Checkpoint Logging:** After finishing the particular data, a checkpoint is written. In case of a failure the checkpoint can be used as an entry point.

Rule 7: Partition and Scale Applications Automatically

> *The seventh requirement is to have the capability to distribute processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.*

This rule implies scaling according to the changing workload without user interaction. At this point it is not possible to do this with Apache Spark. Apache Spark does not provide any features to do that on its own.
As Apache Spark is a framework that is developed for distributed computing it is not an issue, a high-level API is given so the necessary features could be implemented in a monitoring application. The workers can run on several machines without any consequences.

Rule 8: Process and Respond Instantaneously

> *The eighth requirement is that a stream processing system must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.*

The benefits in the RDD and its direct acyclic graph are performance due to high level optimization. Subjectively it is probably more efficent than other concepts in terms of load and overhead.

Conclusion To conclude, Apache Spark Streaming fullfills the requirements for a data stream management system. Although it was not build with the requirements in mind, they are all mostly covered by the product. The only feature that needs improvement and perfection is the automatic scalability of the product.

2.4 Java Example

With a simple wordcount application in Java the concepts of Apache Spark can be illustrated. The implementation follows the map reduce pattern but is just an illustration how RDD works. RDD-Actions enable the developer many more ways to use highlevel querries to perform tasks.

```
JavaRDD<String> textFile = sc.textFile("...");

JavaRDD<String> words = textFile.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s) {
      return Arrays.asList(s.split(" "));
    }
  }
);

JavaPairRDD<String, Integer> pairs = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String s) {
      return new Tuple2<String, Integer>(s, 1);
    }
  }
);

JavaPairRDD<String, Integer> counts = pairs.reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) {
      return a + b;
    }
  }
);

counts.saveAsTextFile("...");
```

- **Line 1:** Initialy the textfile is read into a JavaRDD object, which is the Java abstraction of the RDD.
- **Line 3-9:** The String is split by the blank char " ". The result is then then a JavaRDD List *words*
- **Line 11-17:** The list of words is split into a touple with every word as the key and 1 as its value.
- **Line 19-25:** The touples are now reduced to a single number with the reducedByKey function.
- **Line 27:** Saves the result to a file...

3 Implementation

After analyzing and explaining the concepts of this product a sample implementation for a few given queries was made. At a later stage a benchmark was defined during the seminar to make it 'comparable' between different products.
At this point it is important to mention that not all participants used the same machine nor environment to run the queries, therefore a direct comparison should be avoided!

Exercise To test and implement a comparable case it was decided to use an OSM based Kafka stream. Samuel Kurath provided a blackbox that generated a Kafka stream out of all the OSM updates that are happening during each minute. This incoming Kafka stream should be processed and a defined set of queries should be applied to the data set.

- Leader board of top 10 OSM active users
- Leader board of top 10 OSM objects added
- Node objects with suspicious keys and values
- Way objects with only user tag "area=yes" without other user tags

The key idea of this exercise is to handle a given Kafka stream given by a blackbox and process it.

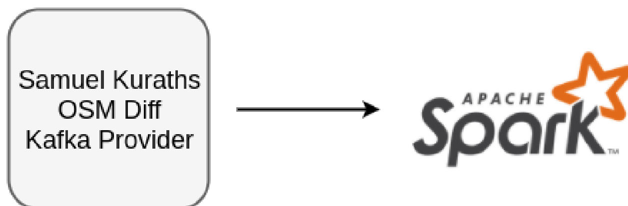

Figure 3.1: Kafka to Spark

Solution The solution was inspired by the Apache Spark provided example code. For data streams the Streaming-Extension was used and for Kafka the Streaming-Kafka-Assembly was used. During the implementation it showed to perform better than the apache-spark-streaming-kafka library as the first one seams to work smoother with the new version 2.x.
The simple concept behind the solution was to take the incoming message and hand it over to an RDD, in each RDD a few steps for processing were made. In each of the examples the Kafka stream was processed using Googles GSON library and the attributes were extracted. In cases were a map-reduce aggregation was made a further reduce step was introduced to optain a result. The results were printed on the console in order to keep things simple.

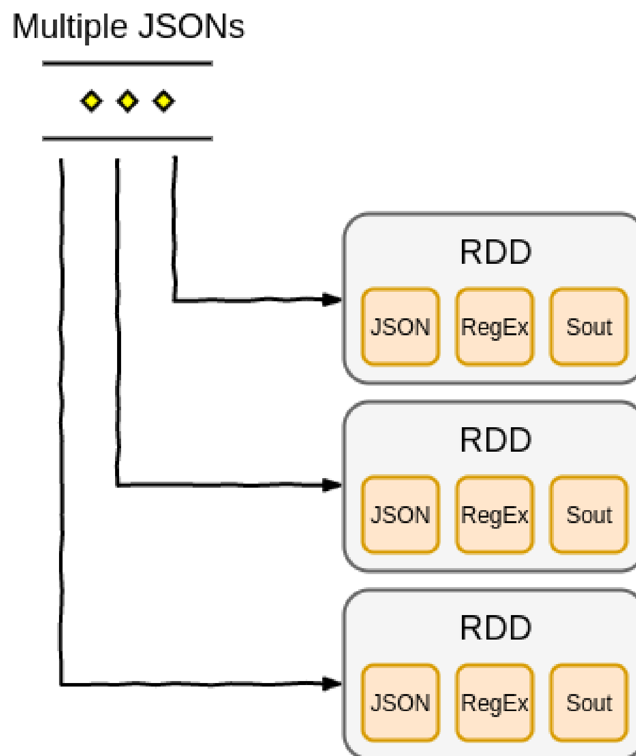

Figure 3.2: JSON to RDD split as a schema

The implementaiton was held simple due to the fact, that the tool chain is very complicated to debug. Apache Spark is very picky with IP numbers and networking. It was shown that the easiest way to run and debug the code is in standalone form on the local machine. Apache Spark has to be insalled locally and ran in a so called master node mode. This mode enables to controll Apache Spark straight out of Java by initializing the parameters directy instead of doing so in a configuration file. The switch to this mode brought stability and repeatability to the environement.

4 Benchmark

The benchmark of the application was made to compare the different products - it should be mentioned, that a direct comparison is very difficult, as not the exact same enviroenement was used between the different participants of the seminar. Nevertheless it should be possible to tell a few things about the trends happening.
To generate the messages all participants were provided by a Python script developed and tested once again by Samuel Kurath. The script is producing messages and pushing it to Kafka. This benchmark is therefore providing an insight on the Kafka consuming performance of Apache Spark!

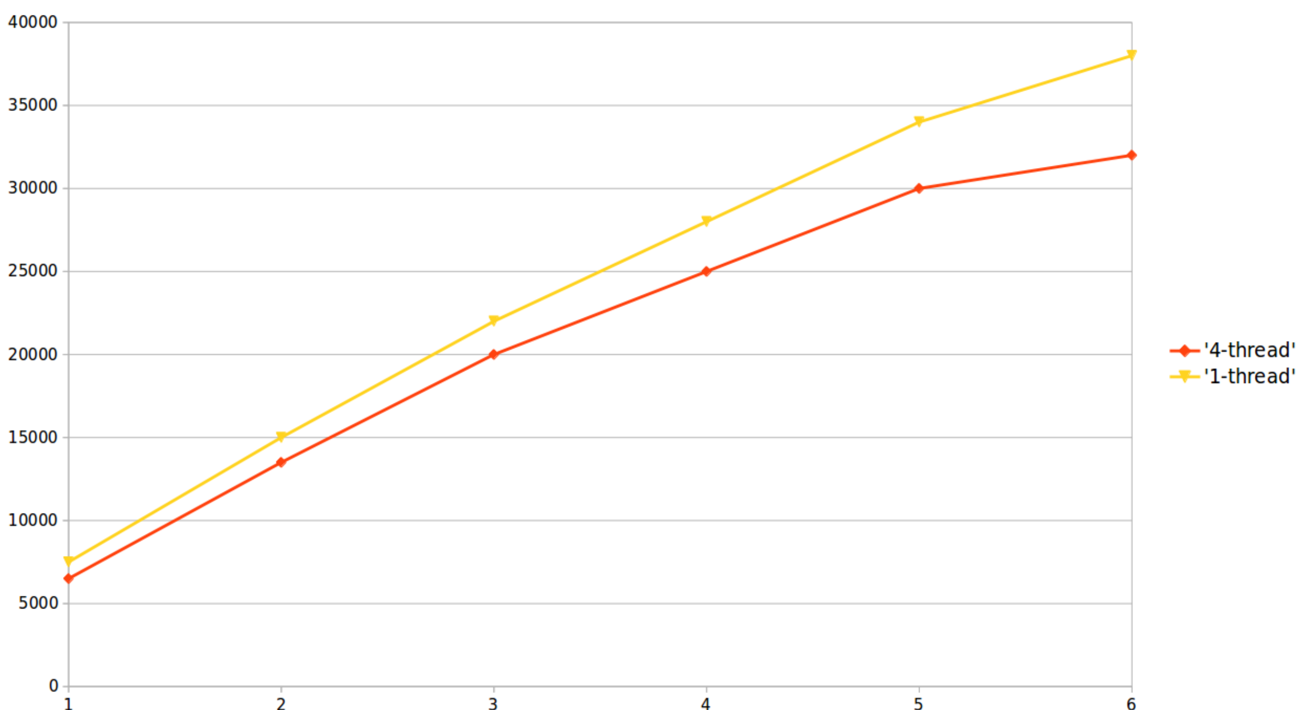

Figure 4.1: Benchmark

The benchmark is showing the computing time in milliseconds over the amount of messages in millions. Apache Spark was run as a standalone master single instance with multiple threads. In order to provide repeatable results the thread numbers were restricted manually. Instead of showing big gains the difference between the two outcomes are neglectable up to 4 million messages. From there the single threaded benchmark starts to stay linearish whereas the 4 threaded benchmark starts to show performance gains. In order to optain this benchmark a series of very precise configurations had to be made. The amount of

threads on the Spark master node had to be limited manually. Due to many failures in the past it is very difficult to say if this one is accurate or not. It is just another random and repeatable benchmark with a particular configuration. At this point it is assumable, that it is representing the processing performance of Apache Spark - but it is very likely that the input is throttled by throughput! In previous versions of the benchmark there was substancial performance gain around the 3 to 4 million messages area, which was random due to the fact, that threading was not configured manually.

Conclusion This benchmark should be taken with a grain of salt. Experience has shown, that benchmarking over this tool chain is very difficult. Although the Kafka container runs on the same machine as the Spark instance it is quite certain that throughput is limiting the benchmark to show the true performance gain optained by more Apache Spark master node threads.
For any further investigation bigger chunks of data should be used and not transfered over any socket but loaded straight from a filesystems which should not be part of the benchmark. There are to many components involved to be certain about the results.

Appendix

Installation Guide

In order to install Spark we need to make sure, that Java 7+ and Scala 2.10.x is up and running.

1. Check Java installation

```
java -version
```

2. Check Scala installation

```
cd ~
mkdir tmp
cd /tmp

# IT HAS TO BE 2.10.xxx
# Kafka will not run on any other version...

wget http://downloads.lightbend.com/scala/2.10.6/scala-2.10.6.tgz
tar xvf scala-2.10.6.tgz

mv scala-2.10.6.tgz /usr/local/scala
export PATH=$PATH:/usr/local/scala/bin
```

3. Install Apache Spark 2.1.0

```
wget http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-hadoop2.7.tgz
tar xvf spark-2.1.0-bin-hadoop2.7.tgz

mv spark-2.1.0-bin-hadoop2.7 /usr/local/spark
export PATH=$PATH:/usr/local/spark/bin
```

4. Verify installation

```
spark-shell

# should return
# scala>..
#
# installation complete!
```

List of Figures

1.1 Transactional Line Item Pattern - to implement an order state 1
1.2 Event Stream - changes can be made visible 2
1.3 CAP-Theorem with its possible overlaps 3
1.4 CRUD operations in a DBMS 4
1.5 Schematic of a DSMS architecture 5

2.1 Apache Sparks and its plugins - Source: Databricks 7
2.2 Apache Sparks internal architecture 8
2.3 Schematic of the map reduce pattern 9
2.4 Schematic of RDD based processing 10
2.5 Spark Streaming feeding batch data to Spark Engine 11
2.6 DStream representation of multiple RDD 11
2.7 Fault tolerant logging for fast recovery 13

3.1 Kafka to Spark 16
3.2 JSON to RDD split as a schema 17

4.1 Benchmark 18

Glossar

CRUD

CRUD represents the four operations handled by a relational database. CRUD stands for Create, Read, Update and Delete. 4

DSMS

Data stream managment systems are database systems to handle continuous streams of data.. 1, 10

DStream

A DStream (Discrete Stream) is the Apache Sparks internal representation for the constant flow of data. 10, 11

RDBMS

Find better defintion. ii, 1, 3, 13

RDD

RDD - Resilient Distributed Dataset is Apache Sparks internal data structure to process data in a parallel and efficent way. 10, 12–16, 19

SparkSQL

SparkSQL is a seamless mix of 'SQL-style' queries and program parts running on Spark. 11

Bibliography

[1] Apache spark 2.1.0 documentation. http://spark.apache.org/docs/latest/, 2017. [Online; 2017-01-18].

[2] Tathagata Das. Spark streaming: What is it and who's using it? https://www.datanami.com/2015/10/05/how-uber-uses-spark-and-hadoop-to-optimize-customer-experience/, 2015. [Online; 2016-09-10].

[3] The Apache Software Foundation. Spark incubation status. http://incubator.apache.org/projects/spark.html, 2017. [Online; 2017-01-07].

[4] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[5] Srikumar S. Rao. Diaper-beer syndrome. http://www.forbes.com/forbes/1998/0406/6107128a.html, 1998. [Online; 2017-01-07].

[6] Hans Rudin. Musterlösung se1 uebung 3: Domain model kinoreservierung. Musterlösung SE1 übung 3, 2013. [Personal Storage].

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