technische universität
dortmund

Masterarbeit

Distributed Stream Processing with the Intention of Mining

Alexey Egorov
February 1, 2017

Gutachter:
Prof. Dr. Katharina Morik
Dr. Christian Bockermann

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für künstliche Intelligenz LS8
http://www-ai.cs.uni-dortmund.de/

Dedicated to Zhanna, Sergey,
Evgeny, Rimma and Geoffrey.

Acknowledgement

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this master's thesis. I express my warm thanks to Prof. Dr. Katharina Morik and Dr. Christian Bockermann for their support, constructive criticism and guidance. They helped me to accomplish this work and steered me in the right direction whenever this was required.

I would also like to acknowledge the proofreaders Leonard, Kai, Lennard, Stefan, Roland and Sebastian for their comments. Last, but not least I am very grateful that my family and beloved ones supported and cheered me up through this researching and writing process and through my years of study.

Alexey Egorov

Contents

1 Introduction

The analysis of streaming data in the context of Big Data has powered research and development of new software frameworks during the past years. A continuously growing number of embedded systems, different sensors, mobile devices and further tools in our daily lives stream high volumes of data. In most cases, real-time requirements are given to process and analyze the incoming data in a very short period of time. Due to the volume of Big Data, it is essential to process it in real-time without the need to store all of it and be able to react fast to the incoming events.

Restricted computing power of a single machine and the limits of network's data transfer rates become a bottleneck for both, stream and batch processing systems in a traditional single-machine setting. Hence, distributed systems for storing and processing data are built. Hadoop's MapReduce and HDFS cluster system was the leading tool for distributed analysis and storage. Through the past years, a wide range of frameworks has been developed to process especially streaming data. Apache Storm is one of the first frameworks that gained a lot of attention while Yahoo's S4 is the oldest one. Other contenders as Apache Flink, Apache Spark or Apache Samza claim to be fast enough to compete with each other. Applying machine learning approaches to streaming data in distributed systems also raises high interest and advances development of distributed machine learning libraries.

Goals of the Thesis In order to execute a task in distributed processing frameworks, usually a specific job for each of them has to be implemented. In this thesis, the aim is to add an abstraction layer for modeling the same tasks using different distributed processing engines. For this purpose the `streams` framework [1] is used to design the tasks decoupled from an engine. It has been developed at the computer science department at TU Dortmund with the intention of working with streaming data sources. While there is a working prototype of modeling jobs for Apache Storm through `streams`, during this thesis the support for modeling jobs using Apache Flink and Apache Spark is added. The core part of this thesis is thus to model existing streaming usecases at such an abstraction layer and compare the performance of the three platforms for distributed computations. Different processing pipelines can be used to evaluate the performance. The analysis of text stream data (e.g. twitter stream) can be performed. A widespread task to compare the throughput rates is counting the words and calculating various statistics. The focus of this thesis lies on more intense processing of complex data streams. Therefore, the image data stream from the FACT telescope is used as a second pipeline to evaluate the different frameworks. It is of great interest to distinguish the better performing distributed processing framework and to understand whether the relative throughput differences between the various usecases can be detected.

Besides performance comparison distributed machine learning libraries and their integration into distributed systems when using `streams` as an abstraction layer is examined basically.

Inspecting the wide range of different machine learning libraries with all the possible combinations does not fit into this thesis completely. However, a short overview over the existing frameworks for distributed machine learning libraries is given. Additionally, a basic comparison of the available machine learning algorithms for each of the distributed processing platforms when using these libraries is performed.

2 Distributed Stream Processing

The estimated amount of data generated by people rises exponentially. The amount of data captured in 2011 at the level of around 1.8 ZB [2] increased to 4.4 ZB in 2013 [3]. At the moment the estimated values for 2020 and 2025 are 44 ZB [4] and 180 ZB [5] respectively. Figure 1 illustrates the growth by the relative size of the circles. As an example Facebook gathered a lot of data and used to process 500 TB of new data per day in 2012 [6]. This development raised the discussion about Big Data. Basically, Big Data refers to data that is too big or too complex to process on a single machine [7]. The most known definition, namely *3V*, defines Big Data using three keywords [8]:

- high **volume** of the data coming from multiple sources
- the formats of the data are different (**variety**)
- the response to a queried data is awaited in real-time (**velocity**)

Though the analysis of Big Data is challenging due to its volume and complexity, it may lead companies to the improvements of their business performance. In case of companies such as Netflix, their recommendation system helps to make better movie suggestions to a person based on the watched movies with patterns learned from the other user groups additionally [9]. Therefore, in this section the way to highly parallelize Big Data processing, the motivation and some issues behind it are considered. Afterwards, several frameworks for distributed Big Data and stream processing are introduced.

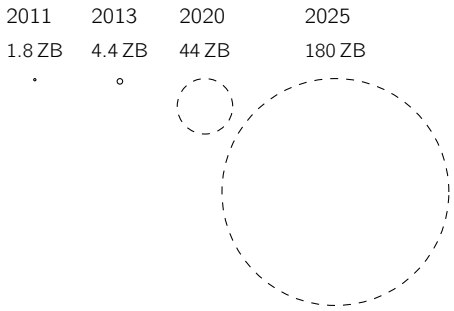


Figure 1: Relative size of the worldwide data

Batch and Stream Processing An important aspect for Big Data is the processing mode which depends on the type of the data and further constraints. Considering a situation with a lot of user logs as mentioned above, the computation task may be calculating reports or other statistics at the end of the day. To achieve this, a series of functions is executed on a batch of data items. Such a scenario is called **batch processing**. The job may be time consuming as all the historical data need to be reviewed.

In the last decade, there was a shift towards Internet of Things with various interconnected devices that produce pieces of information continuously. With the growing number of such devices, the amount of generated data increases. Hence, batch processing is pushed to its limits due to real-time constraints requiring high velocity in terms of data processing. From this setup emerged data streams and **stream processing**. In contrast to batch processing, in

a streaming application the full data set is not available for the processing functions directly at the start. The application receives data items from a source and can only rely on the information collected so far. Furthermore, the source may deliver an unbounded number of data items and the application does not terminate in such a case. Stream processing application is limited by time and space used by the applied functions. The existing algorithms that are applicable to batches require modifications for the streaming environment.

High-Order Operations Recently modern imperative and object-oriented programming languages adopt features that are specific to functional programming. One concept that is also utilized for some of the distributed stream and batch processing frameworks are high-order functions. That is, a function that accepts one or more functions and returns a result value or another function is called *high-order function*. Such functions avoid side effects (e.g. alter global or static variable) and keep data immutable. This means that the functions operate only with the knowledge given by the passed variables. Instead of changing the values of a data structure, a new one is created which is filled with the resulting values. During this thesis the following operations are mentioned:

- *Map* applies a function to each element of a list and returns a list of results: $map(func : (T) \rightarrow U) \rightarrow U[]$. The following code snippet sketches a map function:

  ```
  input:  {"A␣B", "C␣D"}.map(x => x.split("␣"))
  output: {{"A", "B"}, {"C", "D"}}
  ```

  The number of input and output elements is equal.

- *FlatMap* is similar to the above *map* function whereas the result is a flat list of elements. After the map function is applied, the results are concatenated into a list. The following code snippet sketches a flatMap function:

  ```
  input:  {"A␣B", "C␣D"}.flatMap(x => x.split("␣"))
  output: {"A", "B", "C", "D"}
  ```

  In contrast to a *map* function the result is flattened to a single list.

- *Reduce* combines the values of an array. This is accomplished by a function which combines a values with an existing accumulator: $reduce(func : (U, T) \rightarrow U) \rightarrow U$. The following code snippet sketches a reduce function:

  ```
  input:  {1,2,3}.reduce((accum, n) => accum + n)
  output: 6
  ```

  The list of integer values is summed up to a single value.

- *Filter* performs a selection on the input elements which pass a certain condition. The following code snippet sketches a *filter* function:

```
input:  {1,2,3,4,5}.filter(x => x % 2 != 0)
output: {1,3,5}
```

2.1 Distributed Processing Systems

High-volume data coming from different sources needs to be processed under strict constraints of space and time. While sometimes one single physical machine can not process all the incoming data with low latency due to the complex computation steps, in other cases the vast amount of data coming in one time unit exceeds the limits of the network traffic bounds. This urged the idea of splitting computations onto multiple machines and hence building a distributed processing system. In such a distributed system, several shared-nothing machines are combined to a cluster. The machines that work together on a task need to share the data and the results of the computations. Therefore, it is essential to provide those machines with a common network communication. Furthermore, distribution of a task to be calculated, load sharing and allocation of the resources need to be controlled [10]. A master node that is aware of the computing nodes can accomplish these. A user can then submit a computation job to this system by deploying its job to the master node which then automatically distributes and parallelizes the execution.

Distributed Batch Processing One of the first concepts for processing Big Data in a distributed manner, *MapReduce*, was introduced 2004 by Google [11]. It is based on three steps: map, shuffle and reduce, where map and reduce apply user-defined functions. In the *map* step an arbitrary function is applied on the data set of *(key, value)* pairs which produces some intermediate results:

$$(k, v) \rightarrow [(l_1, w_1) \dots (l_p, w_n)]$$

where $\{w_1, w_2 \dots\} \in W$ is the set of all n output value elements. This way, the result of a single computation w_i is assigned a new key l_j with $1 \leq j \leq p$ while the upper limit for different keys is $p \leq n$. Those keys are used then to shuffle and collect pairs with the same key and provide them to the next phase.

For the *reduce* step, the shuffled results of all *map* functions are processed (e.g. count words). Each *reduce* instance gets the data grouped by the keys (e.g. some certain word is a key while counting words of a given text):

$$(l_j, [w_{i_1}, w_{i_2}, \dots]) \rightarrow x_i$$

where a subset of values $\{w_{i_1}, w_{i_2}, \dots\} \in W_i$ with $W_i \subseteq W$ has the same key l_j and hence the function produces the end result x_i. The advantage of this concept is that all the steps can be computed in a distributed system in parallel.

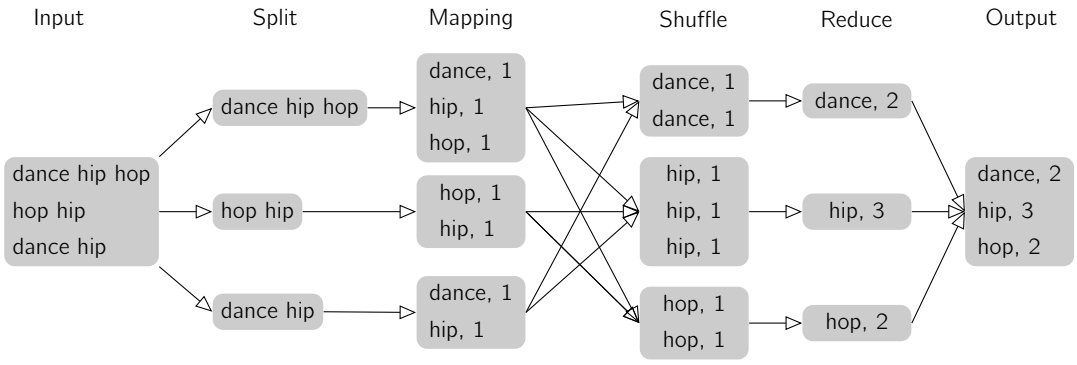


Figure 2: A word-count example for a MapReduce job

Figure 2 illustrates counting words graphically. The data is first split over three computing nodes where each applies a user defined *map* function. After that, all emitted pairs are shuffled by the keys before the *reduce* function performs the count operation over the items with the same key.

Distributed Stream Processing Since the introduction of MapReduce, a bunch of new distributed processing frameworks has been developed and gained vast demand. While MapReduce improved processing batches of data, it was lacking the ability to process real-time data and to deliver quick responses at the same time. Therefore, these novel frameworks share requirements for real-time stream processing. Stonebraker et al. [12] described these requirements in 2005. Most of them are still essential for distributed stream processing. A streaming application should not have a need to store the data itself, but be able to store some information e.g. for stateful processing. Such an application should be simple to distribute over multiple physical machines while delivering low latencies and staying highly available. Of course, it is important that the results of any stream application are replicable and predictable. Furthermore, the authors require handling of stream imperfections (e.g. out-of-order) and the support for some high-level language as SQL for querying data. During this thesis the last two aspects are not discussed deeply to focus on the other requirements for a distributed stream processing system. In section 4.1, some high-level languages similar to SQL are presented to query and process distributed data.

In the following, first the issues of any distributed system and conceptual ways to solve them are discussed. Then criteria for the comparison of distributed stream processing frameworks are defined and after that three frameworks are introduced.

2.2 Lambda and Kappa Architectures

Various frameworks for distributed data processing have been introduced in the past few years. Each distributed system aims at achieving high throughput whereas the performance

gain comes at the cost of possible machine failures and hence data loss. In the following the CAP theorem that describes the issues of building a distributed system is presented. Afterwards two conceptual architectures are introduced which state to solve the issues of the CAP theorem.

CAP Theorem In 2000, Brewer spoke at PODC (Principles of Distributed Computing) about the complexity and issues of distributed systems [13]. In terms of such systems, he suggested to consider three core concepts:

C consistency: all the data across multiple machines is consistent and all requests to the system achieve the same response

A availability: every request must result in a response

P partition tolerance: the system continuous to work, even if some connections between the machines fail (partitions)

Following the CAP theorem, any distributed system aims at holding all three of these ideas. Brewer postulates that it is impossible to provide all of the characteristics, but to combine pairs as 2 out of 3: CA, CP or AP. Hence, it can be represented as a triangle. 12 years later, Brewer corrected his statement himself. Although the probability of a single machine to fail is very small, considering a big cluster with hundreds of the machines partitioning becomes possible. The probability $P(\text{any failure})$ of any physical node to fail and split the cluster into one or more partitions is given as:

$$P(\text{any failure})_n = 1 - P(\text{individual node not failing})^n$$

where n is the number of nodes. Assuming the probability of a node to not fail is rather high, the increasing number of computing nodes still leads to an increasing probability of partitioning. With $P(\text{individual node not failing}) = 0.999$ and $n = \{50, 100\}$:

$$P(\text{any failure})_{50} = 1 - 0.999^{50} = 0.0488 \quad \approx 5\%$$
$$P(\text{any failure})_{100} = 1 - 0.999^{100} = 0.0952 \quad \approx 10\%$$

Thus, ignoring partition tolerance (P) and choosing the combination CA is not applicable [14]. From here on two types of distributed systems are possible as illustrated in Figure 3. Considering an AP oriented distributed system, it is always available even though the computing nodes are partitioned. That is, same requests may not deliver consistent results due to possibly failed connections between some machines in the cluster. On the other hand, if one chooses to build a CP oriented distributed system the solution is to block every request during some failing computing nodes in order to preserve the data consistency.

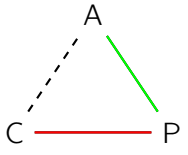


Figure 3: Illustration of CAP theorem

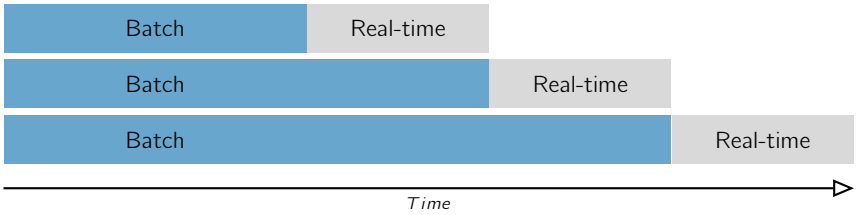


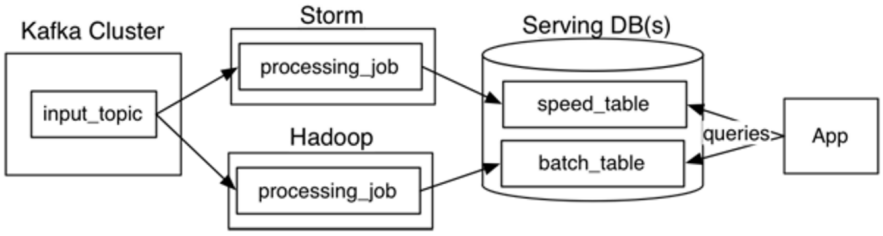
Figure 4: For Lambda Architecture speed layer processes the real-time requests in the time slots between the last and the coming batch reprocessing.

Lambda Architecture As discussed above, a distributed system has to choose between availability and consistency. Let us consider an AP oriented system as in most cases (e.g. like social media such as Facebook or Twitter) the service has to be available. Every query get its result from a precomputed model. For this a batch process is used to process the data. Under the assumption that after the batch process finished, no further changes are applied to the data set, the response is precise. This means that between the last and the next recomputing step the system returns not entirely correct results. Hence, the missing link is to integrate the current changes. Nathan Marz, the developer of Apache Storm and a long-time user of Hadoop's MapReduce, suggested an architecture with a tradeoff between consistency, availability and partition tolerance by reducing the complexity of the CAP theorem, namely *Lambda Architecture* [15]. The basic idea is to allow high availability of the system at the cost of minor inconsistency or incorrectness in the responses. The concept consists of three layers:

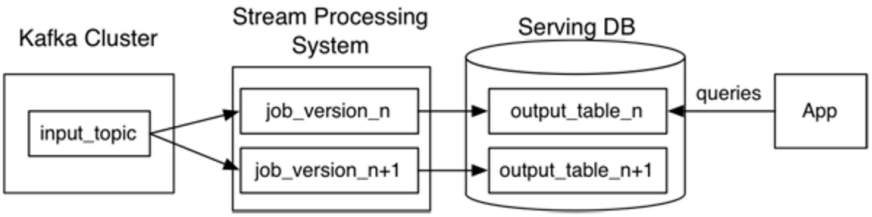
- the speed layer processes real-time events (e.g. using Storm);
- the batch layer reprocesses historical data more precisely (e.g. using Hadoop's MapReduce);
- the serving layer combines and delivers the results of both speed and batch layer to user requests.

As shown in Figure 4, historical data can be reprocessed in batches from time to time whereas the speed layer is responsible to include the data that was not considered during the last batch process. The batch layer restores data consistency and correctes corrupted or wrong responses by using all historical data collected so far. The usage of the speed layer allows real-time events to be integrated quickly into the responses. Figure 5a illustrates the structure of Lambda architecture.

As one of the core concepts, Marz suggests to only work with *immutable data* making data corruption almost impossible. In case of any bugs in the algorithms or other wrong computations, the process only has to be fixed and recompute the data.



(a) Overview of Lambda Architecture



(b) Overview of Kappa Architecture

Figure 5: Outline of the Lambda and Kappa architectures [16]

Some projects as Summingbird[1] or Lambdoop[2] introduced software frameworks for developing applications directly based on the Lambda architecture.

Kappa Architecture The disadvantage of the Lambda Architecture is the use of two different frameworks for the batch and the speed layer. This leads to the following:

- the user needs to program and maintain jobs in two different frameworks
- results from both frameworks have to be merged

Jay Kreps claims that nowadays streaming frameworks are capable of handling both realtime and batch processing. Hence, he introduced the Kappa Architecture as an improvement of the Lambda architecture [16]. The Kappa Architecture simplifies the Lambda Architecture by using a single framework for the batch and the speed layer. It allows development, testing and debugging on top of a single framework. The same code is used for recomputing the batch results and processing real-time requests. As displayed in Figure 5b, a new job running with higher parallelism and more resources is started to reprocess the data. As a side effect, A/B testing can be applied after the recomputing is done because a new second model or database is produced. Thus, the quality of both versions can be verified. As an example A/B testing is applied at Spotify [17]. They state to run newer versions next to the old one while inspecting new performance and statistics values. However, Spotify still relies on the Lambda Architecture[3].

[1] `https://github.com/twitter/summingbird`
[2] `https://novelti.io/lambdoop/`
[3] `http://de.slideshare.net/eshvk/spotifys-music-recommendations-lambda-architecture`

2.3 Requirements and Feature Comparison

All frameworks for distributed computations discussed in this thesis accomplish the same task with minor differences. Each of them claim to achieve higher throughput rates than the others. The engines that are responsible for job execution and their throughput rates are evaluated in section 6. Though characteristics of each framework form the preliminary impression. The first comparison is done by studying the following features of the frameworks:

- *Message delivery guarantee* states how precise is the communication mechanism between the computing instances. Three reliability levels are distinguished:
  - exactly-once: every message is delivered once
  - at-least-once: in case of a failure some messages may be delivered twice or more, with no message gets lost
  - at-most-once: in case of a failure some messages may not be delivered at all
- *Latency* measures the time consumed for processing. It is mostly computed for a computing node or the whole processing tree. The ability to response fast to the queries requires low latencies.
- *Fault tolerance* is provided by frameworks for the case where e.g. a computing node crashes; the items to be processed are restored and replayed by the source
- *Stateful processing* allows for saving the state of an operator and the whole task to a distributed storage. It is used for recovering from a failure and sharing the state of job progress.
- *In-memory* computations are a novel approach to reduce reading from and writing to slow hard drives.
- Support for *iterations* is not a core feature for distributed stream processing applications, but it is required for many batch applications such as machine learning.

Comparison of Spark, Storm and Flink in terms of the properties mentioned above is shown in Table 1. Spark and Flink support in-memory computations and iterative computations without any additional programming needed. They also enable stateful processing, fault tolerance and can guarantee exactly-once message delivery. Storm supports almost the same set of features except for exactly-once message delivery and iterations. Storm and Flink both promise latencies in a sub-second level, while Spark as a batch framework (Spark Streaming works in micro-batch mode) seems to be slower. As these features are advertised by the developers this way, it is misleading as some of them require further preparation and programming and are not available in all situations. More insight is given in section 5.

	Storm	Spark Streaming	Flink
Message delivery guarantee*	at-least-once	exactly-once	exactly-once
Latency	sub-second	seconds	sub-second
Fault tolerance*	yes	yes	yes
Stateful processing*	yes	yes	yes
In-memory	yes	yes	yes
Iterations	no	yes	yes

Table 1: Comparison of distributed processing frameworks. *: implementation or further tuning is required.

2.4 Distributed Open Source Streaming Platforms

With fast evolving technology the number of data sources increased and raised Big Data issues to a higher level. In a distributed batch processing system like Hadoop's MapReduce the data is saved to a file system and processed all at once. Although data storage gets cheaper, reading and writing stays a bottleneck and hence makes such a batch processing framework slow and obsolete.

As discussed above real-time stream processing systems do not require storing the incoming data to process it. Each data item is processed directly after it was received. By filtering and preprocessing the data one is able to reduce even more the amount of data for further stream analysis (feature extraction, data mining, ...). Of course, the data can still be stored for later batch processing. In order to reduce the storage accesses and response time of the system, within the continuous research existing batch algorithms are adjusted for streaming scenarios.

Nevertheless, in many applications the amount of data is still so high, that it can not be stored or processed on a single machine under given time and resource constraints. Therefore, in this chapter we will give an overview over multiple distributed open-source streaming platforms that enable parallelizing their execution over huge clusters of computing machines and improving fault-tolerance to reduce data loss.

All of the frameworks presented here use the same concept to define the jobs that can be distributed and parallelized, namely data flow graphs, that are discussed shortly at first. After that using those graphs the different ways of parallelizing an application are shown. Using some further optimization hints the system can achieve even a higher throughput performance. Then three frameworks are presented here, Storm, Spark and Flink, while a short comparison of them is given at the end of this section.

Data Flow Graph All of the frameworks that are introduced in this section use directed acyclic graphs, DAG, for the definition of the jobs. The nodes represent the sources, tasks

and sinks of the streaming application. All the user logic to analyze and modify the data is placed into these vertices. The edges define the order of the tasks and data movement. Hence, in every job the data flows through the graph from one or more stream sources (a node only with outgoing edges) to one or more sinks (a node only with incoming edges) by passing zero or more tasks (nodes with incoming and outcoming edges). No cycles are allowed to ensure that the data item is processed according to a plan that terminates. Thus, the concept of data flow graphs enables graphical representation of the job and displays how the data flows from one task to another.

Data, Task and Pipeline Parallelism Performance of an application depends on various factors. Although the speed of an application can be increased by more powerful hardware, it was discussed above that the volume, velocity and variety of Big Data pushes hardware improvements to the limit. Therefore, the other possibility is to change the algorithmic concepts of data processing.

The first approach is called **data parallelism** where the same operation is executed on the subsets of the whole data set. Its development started as an opposite to control-parallel algorithms (e.g. multi-threaded tasks) in the late 80s [18]. Data is given in a matrix form where a row is an instance with the attribute values:

$$\begin{matrix} x_{1,1} & x_{1,2} & \dots & x_{1,p} \\ & & \dots & \\ x_{n,1} & x_{n,2} & \dots & x_{n,p} \end{matrix}$$

where n is the number of the data instances (rows) and p is the number of attributes or features (columns). Based on the data structure it can be split horizontally or vertically for improving the parallelism.[4] The data parallelism is exploited in the case of applying machine learning algorithms that are introduced in section 3. Therefore, in the following the two types of data parallelism are explained considering the case of training a model:

- **Horizontal parallelism** splits the incoming data by the features. Each component has a local model that is trained with a streaming algorithm. Periodically those components update the global model. This approach is suitable especially for the case of a high arrival rate of examples. A disadvantage is that every model in each component needs memory for computations. Additionally propagating the model updates is a rather complex task.

- **Vertical parallelism** splits the data by attributes. Each incoming data item is partitioned based on its attributes and then those partitions are distributed over the parallel components. Every component computes statistics and periodically updates the global model.

[4] `http://www.otnira.com/2013/08/03/parallelism-for-dsml/`

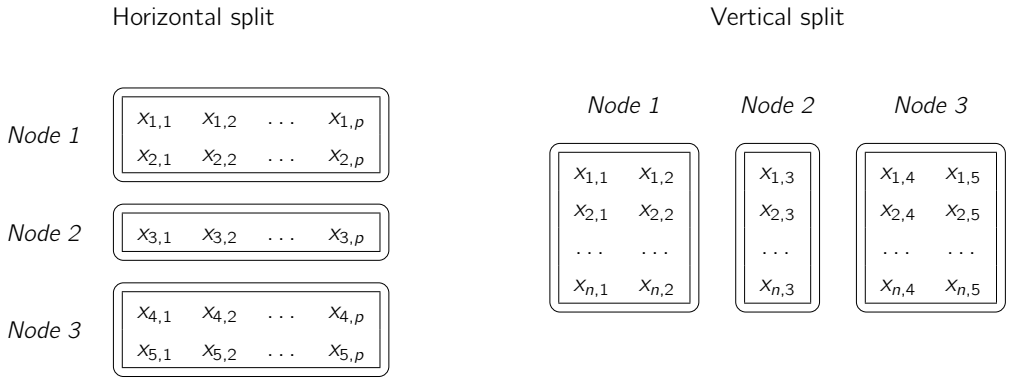


Figure 6: Splitting data horizontally or vertically to distribute the splits over multiple computing machines.

(a) Data parallel: two instances of the same function process two data items in parallel.

(b) Task parallel: two different functions process the same data item in parallel.

Figure 7: Parallelism

An example of splitting the data horizontally or vertically is presented in Figure 6. On the left, each node receives several data rows with values of all attributes while on the right a processing node only receives several data columns with all values of those attributes. Additionally, Figure 7a illustrates how the incoming data items are split for two instances of the same function and processed in parallel.

The second approach is to execute multiple operations or tasks in parallel on the data, **task parallelism**. That is, the algorithm is split into several subtasks that can run simultaneously. Each function is executed independently on the same or different data in parallel. Hence, task parallelism parallelizes e.g. tree nodes of a decision tree over the distributed components. On the other hand, calculating statistics such as min, max etc. can also be done in parallel. This example is illustrated in Figure 7b where two different functions are applied in parallel onto two copies of the same data item.

The third type is the **pipeline parallelism**. Here the output of one task is used as the input for the following one. Each of the pipelined tasks runs in parallel and the latency of the system depends on the task with longest computation.

In real-world applications and frameworks these different types of parallelism are combined. With the pipeline parallelism for a single task of the pipeline multiple instances can be executed to additionally parallelize intense computations.[5]

Optimizations The job graph simplifies not only the job definition, but also further performance optimizations. Hirzel et al. [19] summarizes proposals for stream processing optimizations of the last decade. In following, some of the suggested approaches to improve the performance of the distributed stream processing job are mentioned and discussed shortly.

Job Graph Optimizations As a very first step, the job graph should be optimized by the application expert itself. First, the operations to be done can be reordered in such a way that filtering is done as early as possible. That is, less data items need to be forwarded through the process and hence less computations need to be done. Secondly, in a job where the data is split and doubled into two or more new subgraphs any redundant operations should be reduced.As the last step, the application expert has the knowledge about the computational complexity of the job and is able to support grouping some computationally expensive and cheap operations together. This way this knowledge may be used later to place the groups of operations on to different hosts (CPUs, machines).

Parallelization As it was introduced in the paragraph above, parallelizing the computation can be achieved in several ways. Therefore, the one also suggested in [19] is to increase the data parallelism for computationally expensive operations. That is, the operator is replicated and each instance can process the data items in parallel. Also splitting complex operations into simpler single operations is preferably as one may use different data parallelism on each single task. Furthermore, modern distributed processing frameworks apply their own job optimizations based on the job graph. To achieve the highest performance, the load for each operator should be balanced. This can be accomplished by grouping the data items by the various criteria (e.g. randomly, grouped by some key, etc.). Although, allowing pipeline parallelism next to data parallelism may improve the system throughput, in some cases it might be beneficial to fuse some operators and remove the pipeline parallelism between them. This is especially the case if the communication costs are higher than the computation costs of the second operator.

Load Shedding The source of a streaming application may not produce continuously the same amount of data items per time unit. In many cases the workloads come bursty and the data rate fluctuates over time with the originally high input rates.[6] If one considers

[5] http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concurrency-basics.pdf

[6] http://www.cl.cam.ac.uk/~ey204/teaching/ACS/R212_2014_2015/slides/Peter_Streaming_2014.pdf

a sensor network measuring passing cars, it will account higher amount of data in the rush hours. The system that was tuned for lower throughput may not be able to process the incoming data and thus will tend to fail or increase the response times. It is also impractical to setup a system that can handle the input rates during the peaks as they can be many times higher than the general input rates [20, Chapter 7].

A more practical approach is to automatically adapt to the burst. Load shedding is a technique to solve the problem of bursty workloads. Instead of increasing the delay of the system, it is possible to shed some of the load with a minor impact on the result of the process. That is, the accuracy of the system depends on the particular strategy of the load shedding approach as a subset of the data items is dropped when the bottleneck is detected. The general simple load shedding approach is to decide with some probability p whether the tuple can pass or not. Rivetti et al. [21] introduces load-aware shedding, a modern approach particularly for the distributed stream processing frameworks that aims for dropping as few data items as possible. This new approach computes a cost model based on the time needed to process the items and is able to skip those data items which processing is more time consuming.

Backpressure Considering the situation that was introduced for load shedding, there is another technique to overcome bursty workloads without loosing data items, *backpressure*. A backpressure system monitors its components for a case where a processing node is too busy. This can be measured by controlling how full the component's receive buffer is. When the buffer is full, the source of the stream shall stop or slow down and wait for the system to process the current workload. All frameworks introduced in this thesis support backpressure and allow the user to turn this feature on or off when configuring the cluster.

2.4.1 Apache Storm

Apache Storm is a distributed scalable stream processing framework. After it has been open-sourced in 2011 by its developer Nathan Marz, Apache Storm was acquired by Twitter just one year later and used for its real-time analysis pipeline. The combination of low latency and at-least-once semantics made Apache Storm attractive to the various use cases for companies such as Spotify [17], Yahoo [22], Twitter [23] and others.

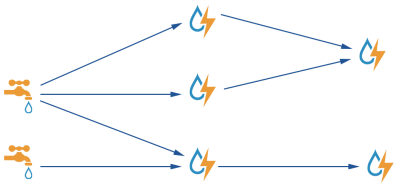


Figure 8: Apache Storm topology showing two spouts spreading incoming streams to subscribing bolts.[7]

Apache Storm's pipeline for data processing, called a *topology*, is a directed acyclic graph (DAG) as shown in Figure 8 that is executed indefinitely until it is killed. Sources of

[7] `http://storm.apache.org/`

data streams in a topology, *spouts*, read an external stream source and produce serializable messages, *tuples*, with an ID and information about its sender and receiver. The processing components, *bolts*, receive tuples from spouts or other bolts, execute some actions on them and emit the results to next bolts. Figure 8 outlines a Storm's topology. Storm distributes the configured spouts and bolts of the topology to the executors. As already mentioned, to improve the performance the load for the bolts should be balanced. Therefore, a strategy of partitioning tuples across the cluster is crucial for many usecases. Storm supports several *groupings*: "shuffle" grouping (equal number of tuples to each task), "fields" grouping (e.g. deliver tuples with the same key entry to the same executor), "all" grouping (replicate each tuple to all executors) and some more.

Reliability is an important point for Apache Storm in order to achieve an at-least-once guarantee level. Storm tracks the flow of tuples through the job graph using internally a tuple tree. This functionality can be enabled by using a special API to acknowledge messages or mark some of them as failed. The data item that reaches the leaves successfully is considered fully processed. Therefore, reliability starts with the spout emitting the tuple with a unique message id which is used to track its state inside a tuple tree. Failed or successfull execution of a bolt's function has to be linked to the tuple tree which is called *anchoring*. For this purpose, the reliability API provides `ack(Tuple item)` and `fail(Tuple item)` methods. When the tuple is fully processed or the processing has failed, the `ack()` or `fail()` API in a spout is called respectively. Without activated acknowledging of the messages, values for latency, acked and failed events will be set to zero. Otherwise Storm works at an at-most-once guarantee level where messages may get lost. Until the state of a tuple is known, it is hold in Storm's memory for a possible replay. Storm allows for tuning the upper time limit for a tuple to run through the topology.

With the introduction of transactional topologies, Storm enables exactly-once message guarantee.[8] This is realized by implementing a topology using `TransactionTopology` API. For this thesis the basic topology API has been used providing at-least-once message guarantee.

API and Trident Storm's low-level API is kept rather simple. Nevertheless, this also means for the user that operations such as *map* or *flatMap* to be applied onto the data stream have to be implemented. Trident is a high-level abstraction on top of Storm. It introduces joins, aggregations and filters to the existing functions and grouping capabilities of Storm. Furthermore, Trident supports stateful processing and exactly-once semantics, but at the cost of slightly higher latencies.

[8] `http://storm.apache.org/releases/1.0.0/Transactional-topologies.html`

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 2);
builder.setBolt("split", new SplitSentence(), 2)
        .setNumTasks(4)
        .shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 3)
        .fieldsGrouping("split", new Fields("word"));
```

Figure 9: Word count example using Apache Storm

As a comparison two code snippets are considered to solve the same "word count" task. First, Figure 9 presents how this is done in Storm. While this code snippet[9] only configures the Storm topology, the actual processing task is hidden in the two classes `SplitSentence` and `WordCount`. They are defined by the user producing an overhead. The task is split into two bolts where the first one is splitting sentences into words and the second one is counting words. Regardless of the implementation of counting, *field grouping* by words is applied here That is, same words will be processed on the same node and thus the computation can be distributed easily.

The code snippet in Figure 10 shows a word count example implemented with Trident.[10] The only missing component here is the implementation of the splitting rule in the `Split` class. Trident on its side automatically computes the most optimal topology and groups some of the called functions into bolts. Hence, in the end Trident creates and runs a Storm topology. The user only needs to think of the processing pipeline itself. As this work is concentrated on distributed stream processing, Trident is not considered for the later implementation and evaluation.

Integration with external systems and other libraries is supported. The list of officially added connectors is shown in Figure 11a.

Parallelism The parallelism of a topology depends on the settings given by a user and the cluster ressources. The coordination of a Storm cluster is handled by the *nimbus* node that distributes a streaming job to the *supervisor* nodes or *workers* (worker machines). Each worker executes a subset of the topology. Each supervisor provides a number of *executors* that should be smaller or equal to the number of the CPU cores. The overall parallelism relates to the number of *executors* or threads that are spread across the workers as it can be seen in Figure 11b. Furthermore, each executor spawns one or more *tasks* for a bolt or spout to be used. The *task* performs the data processing and is run inside of an *executor*'s thread. Therefore, multiple tasks in one executor are executed serially. Hence, mostly it is preferable

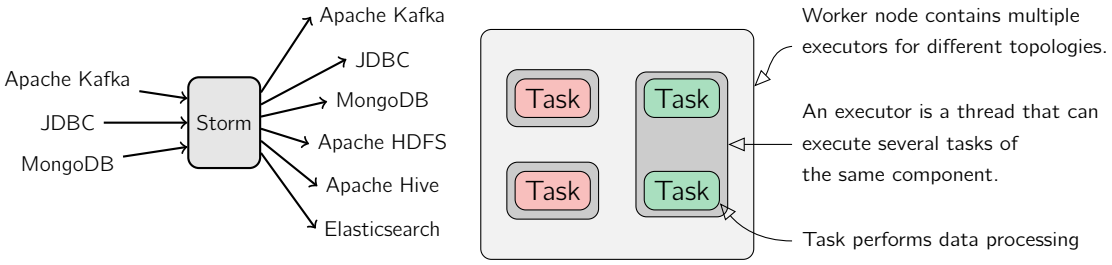
[9] `https://github.com/nathanmarz/storm-starter/blob/master/src/jvm/storm/starter/WordCountTopology.java`

[10] `https://github.com/nathanmarz/storm-starter/blob/master/src/jvm/storm/starter/trident/TridentWordCount.java`

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts = topology.newStream("spout1", spout)
        .parallelismHint(16)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))
        .parallelismHint(16);

topology.newDRPCStream("words", drpc)
        .each(new Fields("args"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .stateQuery(wordCounts, new Fields("word"), new MapGet(), new Fields("count"))
        .each(new Fields("count"), new FilterNull())
        .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

Figure 10: Word count example using Trident



(a) External connectors that are officially supported by Storm.

(b) Apache Storm topology parallelism where each color corresponds to a spout or bolt. Different colors imply different tasks.

Figure 11: Apache Storm: 11a presents connectors for Storm; 11b illustrates components of Storm.

to have only one task per executor and each component (spouts and bolts) is set per default to use one task. In contrast to the number of tasks that can not be changed throughout the lifetime of the topology, the user can change the amount of workers and executors. Setting the number of tasks higher than the number of executors allows for achieving a higher parallelism level after adding more worker nodes and rebalancing the topology.[11]

Graphical interface Apache Storm presents the overview of the cluster and the running jobs in its graphical interface. Directly on the landing page, the user can find a summary about the cluster, active topologies, registered supervisors and the configuration of the nimbus node. A part of the landing page can be viewed in Figure 12.

On the page of a certain topology, one can perform some actions such as *activate*, *deactivate*, *rebalance* or *kill*. Deactivating and activating the topology is like holding and resuming the processing of the data in the topology. Rebalancing can increase the level of

[11] http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/

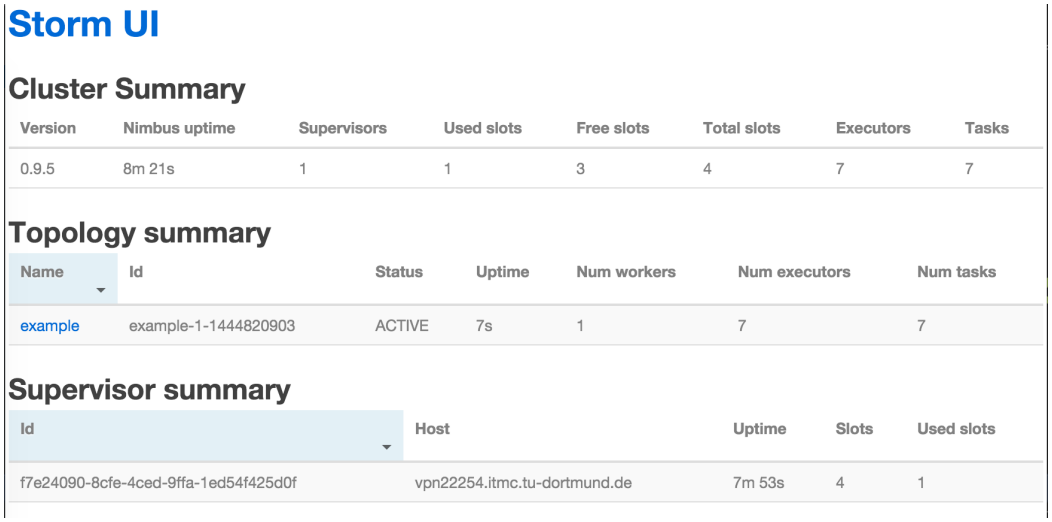


Figure 12: Landing page of Storm's UI

parallelism and improve the throughput rate if more processing nodes become available after the job started and the number of tasks has been set higher than one. Kill command stops the processing and removes the job from Storm. Further statistics about the topology are listed in the table view. As mentioned above, statistics about latency, acked and failed events are available only if the Storm job is implemented using the reliability API. A visualization graph of a topology is also shown and updated live with the information about the thoughput numbers. Additionally, details on spouts and bolts themselves can be viewed on a component page respectively.

Generally, Storm's dashboard presents a lot of information to the user. Some of it is shown in several ways on the different subpages. Especially with activated message-acking Storm helps to understand the data flow.

2.4.2 Apache Spark

Apache Spark is another distributed processing framework based on batch processing [24]. Since its first stable release in 2014 Spark achieved a lot of attention in the last couple of years as a contender to Apache Storm and Hadoop's MapReduce.

Authors of Spark claim to be up to 100x faster than a regular MapReduce job if the computation fits in memory and up to 10x faster otherwise.[12] Databricks proved Spark's performance quality without using its in-memory capabilities [25]. They submitted and won the GraySort test[13] in 2014 for sorting 100 TB [26]. Hadoop MapReduce sorted 102.5 TB in 72 minutes with 2100 nodes, Spark was able to reduce the sorting time to 23 minutes for

[12] `http://spark.apache.org/`
[13] `http://sortbenchmark.org/`

sorting 100 TB with only 206 nodes. This way Spark sorted the same amount of data 3 times faster and used 10 times less computing nodes.

With almost 1000 contributors[14], further libraries have been implemented on top of Spark to enable support for SQL, graph processing, streaming and machine learning. For this thesis the last two libraries, streaming and machine learning, are essential. These enable to additionally process streaming data and apply machine learning algorithms.

Distributed Batch Processing Apache Spark as a distributed batch processing framework was developed around the concept of *resilient distributed dataset* structure (RDD) for the internal memory management of distributed processing frameworks [27]. RDDs are constructed as immutable, cacheable and lazy containers to model more than just Map-Reduce jobs.

Tasks in Spark are built like in Storm as directed acyclic graphs (DAG). The core idea is to define **transformations** and **actions**. Those operations can be applied to many data items of a dataset. Each transformation creates a new RDD that knows its predecessor (parent), but the transformations are not computed immediately. Thus in case of a failure, using the lineage of RDDs, one can replay the previous transformations. More details on failure recovery can be found in section 5.7.2. An operation defined by RDD is executed only when some *action* is called on it. All intermediate results are kept in-memory and are written to disk in the case of low memory or on an explicit user request (`persist` or `cache`).

Additionally, each RDD splits its internal collection of elements into partitions. The number of partitions determines the level of parallelism of the application. Controlling and tuning the partitioning right increases the runtime performance[15]. If any partition is lost, it can be recomputed because of RDD's immutabiliy and the remembered lineage of the transformations. The internal scheduler splits submitted jobs into `stages` where each stage processes one partition of an RDD. A stage combines multiple transformations on a single partition. Whenever the data set is shuffled, a new stage is created. The shuffle operation copies data across various executors including different physical machines in order to distributed it differently across partitions. This makes it a costly operation. Figure 13 illustrates stage building for a Spark job.

Storage Levels Spark allows for various levels of storing RDDs. There are three basic choices to meet:

- *memory only*: the data is kept in-memory only. This is also the default and most CPU efficient setting for Spark, but if a partition does not fit to the memory, it has to be recomputed.

[14] `https://github.com/apache/spark`

[15] `http://www.cs.berkeley.edu/~matei/talks/2012/nsdi_rdds.pdf`

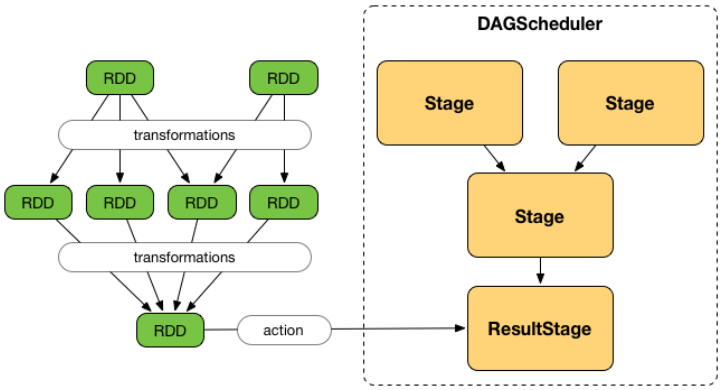


Figure 13: A Spark job is split into stages at the points of DAG when the data is shuffled.[16]

- *memory and disk*: the data is kept in-memory unless the application is running out of memory. In that case some partitions may be stored to the disk.
- *disk only*: all data is stored on the disk and not in memory.

RDD objects can be made more space efficient by serializing them. The first two options that allow memory as a storage support *object serialization* while this means a higher CPU usage to deserialize them. Finally, to improve the reliability of the system each of the above storage options support higher replication. That is, every partition is replicated on two cluster nodes.

Spark Streaming As processing data streams received as much attention as batch processing, the Spark project was extended with streaming capabilities, called *Spark Streaming*. In contrast to Storm, Spark Streaming is not a record-at-a-time system. It *"treats a streaming computation as a series of deterministic batch computations on small time intervals"* [28]. While for batch processing Spark uses the concept of RDD, Spark Streaming is based upon another abstraction above RDD, a discretized stream called *DStream*. A DStream contains a continuous series of RDDs. The data from a certain time intervall is collected into an RDD. Each operation applied on a DStream is translated to an equivalent operation on the underlying RDD object. Figure 14 sketches the described concept.

All possible operations on the DStream can be divided into two groups: **transformations** and **output operations**. Spark Streaming follows the same concept of lazy evaluation as in Spark. In this case it means that no *transformation* is performed unless any *output* operation is applied onto a DStream. The following operations are supported on DStream:

- `transformations`: *map*, *flatMap*, *filter*, *union*, *count*, *reduceByKey*, *transform*, *join*, *window*, *updateStateByKey*, ...

[16] `https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-dagscheduler-stages.html`

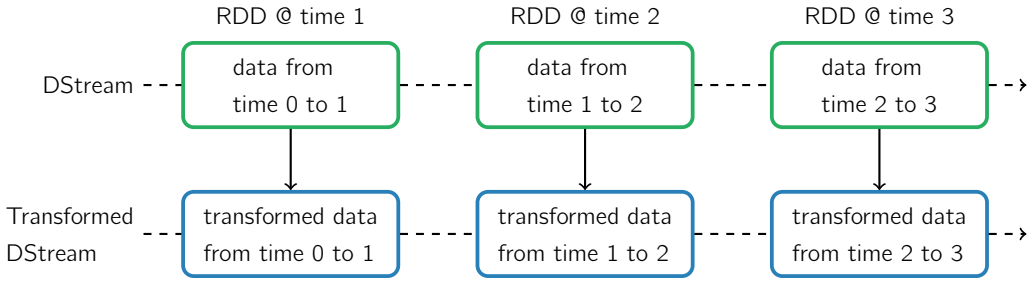


Figure 14: DStream concept is based on RDDs and allows high-level API to perform transformations on the input data stream.

```
private static final Pattern SPACE = Pattern.compile(" ");
JavaReceiverInputDStream<String> lines = ssc.socketTextStream(
            args[0], Integer.parseInt(args[1]), StorageLevels.MEMORY_AND_DISK_SER);
JavaPairDStream<String, Integer> wordCounts =
        lines.flatMap(new FlatMapFunction<String, String>() {
                public Iterator<String> call(String x) {
                        return Arrays.asList(SPACE.split(x)).iterator();
                }})
        .mapToPair(new PairFunction<String, String, Integer>() {
                public Tuple2<String, Integer> call(String s) {
                        return new Tuple2<>(s, 1);
                }})
        .reduceByKey(new Function2<Integer, Integer, Integer>() {
                public Integer call(Integer i1, Integer i2) {
                        return i1 + i2;
                }});
wordCounts.print();
```

Figure 15: Word count example using Spark Streaming

- `output operations`: *print*, *foreachRDD*, *saveAsHadoopFiles*, ...

Many transformations that are used on RDDs are also available on the DStream. Some special transformations are for example `windowed` operations. They allow to apply operations on a sliding window. Spark Streaming can maintain an arbitrary state using operations such as *updateStateByKey*. This way the collected information or statistics are updated continuously. As DStream is also evaluated lazy, output operations force the execution of the transformations. The easiest way is to append a *print* operation after the transformations. Other operations are usually used to persist the resulted RDD locally or to an external storage.

Figure 15 presents a word count example in a Spark Streaming environment. At first, the stream is defined. Then using a *flatMap* function the lines are split into words, *mapToPair* function maps each word to pairs (the word and the counter set to 1) and at the end during the reduce step pairs with the same "key" (words) are counted with the *reduceByKey*

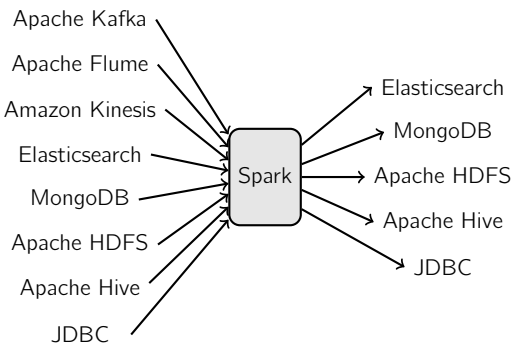


Figure 16: External connectors officially supported by Spark Streaming

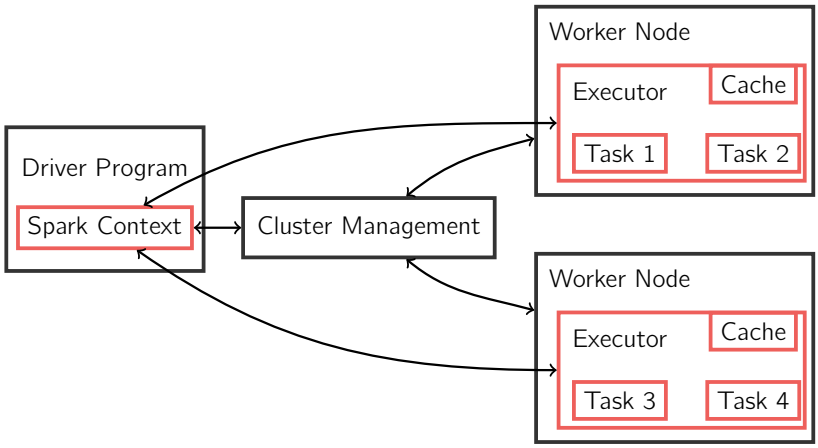


Figure 17: Spark cluster configuration

function. After all those steps, in order to force the execution of the defined job,an output function such as *print* is called.

Spark Streaming supports various data stream sources and sinks. Figure 16 illustrates some of them. With the active community the list of supported external libraries is extending continuously.

Parallelism For Spark it is important to tune the system carefully in order to be able to use all the ressources. A Spark cluster consists of a master and multiple worker nodes. Every worker node is assigned a particular number of executors where the CPU cores available at the worker can be used by the executors. In the cluster setup in Figure 17 the driver program appears as an important actor for cluster management. Using *SparkContext* the user declares the job topology and the needed ressources. The driver programm containing the SparkContext communicates with the *cluster manager*, requests the ressources and distributes the tasks to the executors. Additionally, the results of operations like *reduce* are collected in the driver program and hence should be planned carefully.

As mentioned above, an RDD is a partitioned dataset and Spark Streaming can process multiple partitions of the data in parallel. Following the documentation of Spark and the tuning recommendations by Maas [29] in order to use all cores for the processing the user should tune the system to split the incoming data stream into enough partitions. The number of partitions can be computed easily using the following equation:

$$\#partitions = \#receivers \times batchInterval / blockInterval$$

where $batchInterval$ is a time interval to collect the items from the stream. $blockInterval$ controlls in how many blocks the data from a batch is split. The number of blocks in each batch determines the number of tasks that will be used to process the received data in a map-like transformation. Each task is executed on a single core. There are two parameters to tune, $batchInterval$ and $blockInterval$. The user defines the batch interval. Then the block intervall is computed as the number of receivers and the amount of partitions is known ($\#partitions = \#usableCores = \#allCores - \#receivers$):

$$blockInterval = \frac{\#receivers * batchInterval}{\#partitions}$$

As $batchInterval$ is still a variable that can be chosen by the user, finding the right settings might become a more difficult task than e.g. for Apache Storm.

Even after adjusting the number of partitions right, in some situations it is advisable to force the reduction or increase of the level of parallelism. This can be done by calling `repartition(`*`partitions`*`)`. The pipeline which follows the `repartition` function call, will be then split into a desired number of partitions. Spark allows for achieving the same effect if using the following API calls:

```
groupByKey(partitions)
reduceByKey(partitions)
```

Graphical interface Sparks' graphical interface is an essential tool for collecting information about the running jobs. The UI provides a lot of information about the progress of tasks. In order to tune a Spark job properly one needs to inspect this information. On the front page the user can directly view the status of the current jobs. The menu on the top presents the overview of the different *stages*, *storages*, *environment* and *executors*.

For a current streaming job Spark adds another menu, *Streaming*. While developing a streaming application this is a good start to inspect the performance of the system. As can be seen in Figure 18, the following statistics are presented under the streaming menu:

- *Input rate* shows statistics over all the stream sources and the average throughput of events per second; the user is able to examine the input rate of each source working in parallel;

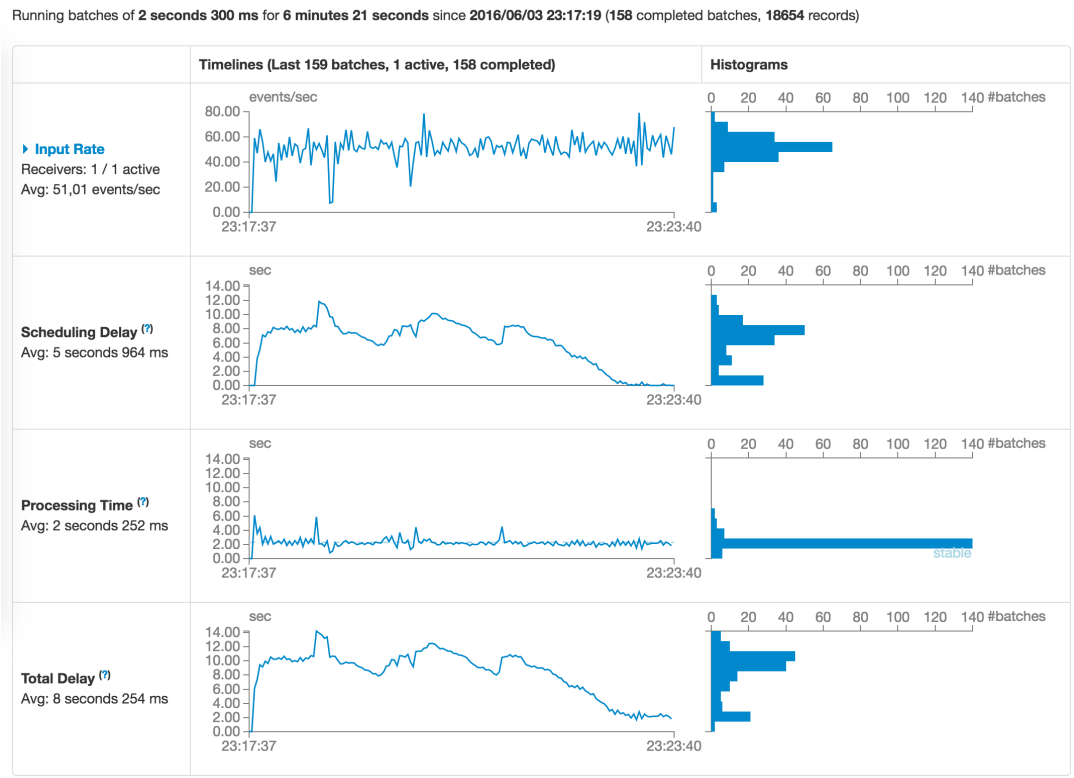


Figure 18: Details of a Spark Streaming job. The system recovers from the initial delays.

- *Scheduling delay* indicates the ability of the system to process the incoming stream data in the given batch interval time;
- *Processing time* shows the time needed for the system to process each micro-batch;
- *Total delay* presents summed *scheduling delay* and the *processing time*; that is, the newest incoming batch will be processed after this delay time;

Further down Spark presents the current queue of batches, active batch processing and already finished batches. In a worst case scenario, the delay is increasing such that the system can not process events in time and the queue of batches is growing. Each processed or active batch can be inspected and the job plan and the stages (failed and finished) can be viewed. It is possible to dive even deeper into the execution times of each single task. This way the user can determine bottlenecks and time consuming components, verify whether all ressources are in use and understand the execution logic of the job.

As already described in the previous paragraph tuning Spark depends on the right partitioning, good fitting batch and block intervals. For tuning purposes and also in case of a failure the additional information obtained from the `storage` or `executors` tabs is useful. `Storage` gives an overview of all RDDs, how much memory is used for them and whether they are saved in memory or on a disk. Especially in case, where the system is not fast

enough, the increasing amount of data to be processed can be determined. Otherwise it might increase until all memory is used up. In the `executors` view Spark shows actively working executors and here worker machines that do not receive any data or not use all of the available resources can be detected. Though this can be noticed in the batch details view.[17]

2.4.3 Apache Flink

The newest contender of Storm and Spark, Apache Flink, gained a lot of attention especially in Europe [30]. Flink started around 2010 (originally: Stratosphere) as a research project funded by the DFG. Towards the end of 2014 it became an Apache project.

Flink is a streaming-oriented engine that executes distributed batch and stream jobs using the same streaming engine as shown in Figure 19b. The batch and stream APIs are almost identical with the basic types `DataSet` and `DataStream` respectively. The single major difference in the API is the call to group the data items by a key with `groupBy()` used for batch and `keyBy()` for streaming applications. `DataSet` API provides access to multiple libraries which adds machine learning, graph processing and other capabilities to batch processing. For `DataStream` API a library for processing complex events is provided and the additional `Table` API allows SQL-like syntax to define Flink jobs.

Features As seen in the overview in section 2.3, Flink supports exactly-once semantics, stateful processing and in-memory computations just like Spark. As it can be seen in Figure 19a Flink supports snapshots to improve fault tolerance. This way in case of a node failure only results since the last snapshot have to be recomputed. More details on fault tolerance can be found in section 5.7.2. Although most features overlap with Spark, minor details differ.

The **pipelining** approach is Flink's feature to avoid the materialization of large intermediate results.[19] In order to achieve this, intermediate results are not materialized, but piped into the next transformations. As an example one can consider the following sketch for a task:

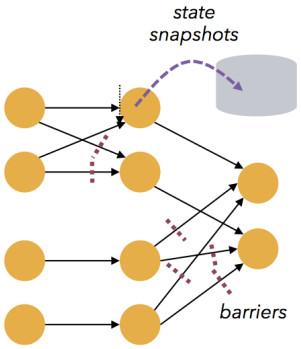
```
DataStream result = stream.map( ... );
DataStream nextResult = result.filter( ... );
nextResult.print()
```

Flink will not materialize `result` and pipes it directly to the *filter* operation. With pipelining the operators start processing as they get the data and do not wait until their predecessors

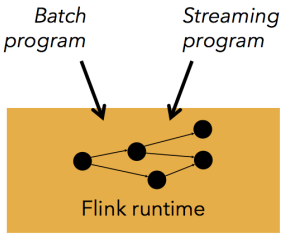
[17] `https://databricks.com/blog/2015/07/08/new-visualizations-for-understanding-apache-spark-streaming-applicat`
`html`

[18] `https://flink.apache.org/features.html`

[19] Slides 18-20 in `http://de.slideshare.net/fhueske/apache-flink`



(a) Distributed snapshots (states) in Flink job



(b) Flink capable of running same job as batch and stream process

Figure 19: Capabilities of Flink's engine[18]

finish the tasks.[20] Thus most of the data is kept in memory. At the same time efficient **memory management** is one of Flink's advertised features. It aims at reducing the garbage collection and hence increases the available computing time.

In contrast to Spark **iterations** over a data set in Flink are made simple with the `iterate` API call. This feature has even been extended to improve performance of iterations. For the so called *delta iterations* feature Flink explores which part of data is changed during the iterations and applies the operations only on that part of data. Iterating over a stream is also possible. This adds Flink another promenint feature not available on other distributed stream processing platforms. Finally, Flink developers emphasize that the optimizer algorithm improves Flink performance for batch jobs. With the above features Flink claims to provide latencies similar to or even better than Apache Storm in a sub-second level in contrast to Spark.

In the code snippet in Figure 20 one can see an example of a Flink program[21]. As a task the word counting example is used. The steps are similar to those in Spark Streaming. First, splitting lines to words is done using the *map* function. Inside the *flatMap* function single words are mapped to pairs. Then the stream is grouped by collected words (parameter number 0) and a sum is built over the second parameter of each pair (parameter number 1). Flink also requires an output operation such as *print* to be defined in order to force the actual execution of the job plan.

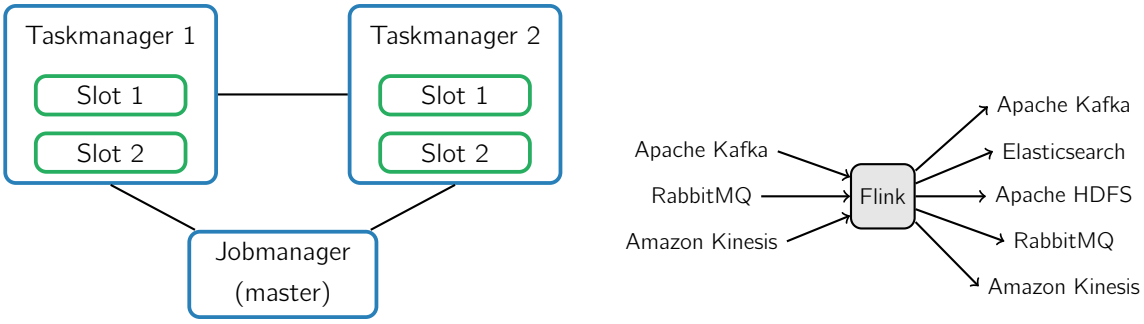
As the previous frameworks Flink also supports external connectors. Figure 21b shows some of them that are advertised by Flink developers themselves.

[20] Slides 29-36 in `http://de.slideshare.net/KostasTzoumas/apache-flink-api-runtime-and-project-roadmap`

[21] `https://github.com/apache/flink/blob/master/flink-java8/src/main/java/org/apache/flink/streaming/examples/java8/wordcount/WordCount.java`

```
DataStream<String> text = getTextDataStream(env);
DataStream<Tuple2<String, Integer>> counts = text
        .map(line -> line.toLowerCase().split("\\W+"))
        .flatMap((String[] tokens, Collector<Tuple2<String, Integer>> out) -> {
                Arrays.stream(tokens).filter(t -> t.length() > 0)
                        .forEach(t -> out.collect(new Tuple2<>(t, 1)));})
        .keyBy(0)
        .sum(1);
counts.print();
```

Figure 20: Word count example using Apache Flink



(a) Sketch of a Flink cluster setup consisting of a master node and multiple taskmanagers

(b) External connectors advertised for Flink

Figure 21: Apache Flink

Parallelism Tuning the level of parallelism in Apache Flink is similar to Apache Storm's concept. As can be seen in Figure 21a, a Flink cluster consists out of two core components: a jobmanager (master node) and (multiple) taskmanagers. The task of a jobmanager is to control the Flink cluster, receive job requests and transmit them to the taskmanagers. Each worker machine is called a *taskmanager*. For each taskmanager the user defines the *number of task slots* that can process the data in parallel before a job can be deployed. This number can not be higher than the number of CPU cores. Every taskmanager can define its own amount of working slots. In the programm definition the processing of the data can be to tuned in more details. In contrast to more complex tuning of Spark in a Flink cluster the simplest way to exploit all the ressources is to set the default level of parallelism to the number of working slots. Furthermore, Flink uses so-called *chaining* to bundle several operators for a single slot. If no instructions are given, Flink will automatically decide which operators will be chained together as a part of the optimization process. Chaining some operators together or implicitly disable chaining for other operators is part of Flink's API. Additionally reserving some computing slots for groups of tasks or sharing slots among all tasks is also possible.

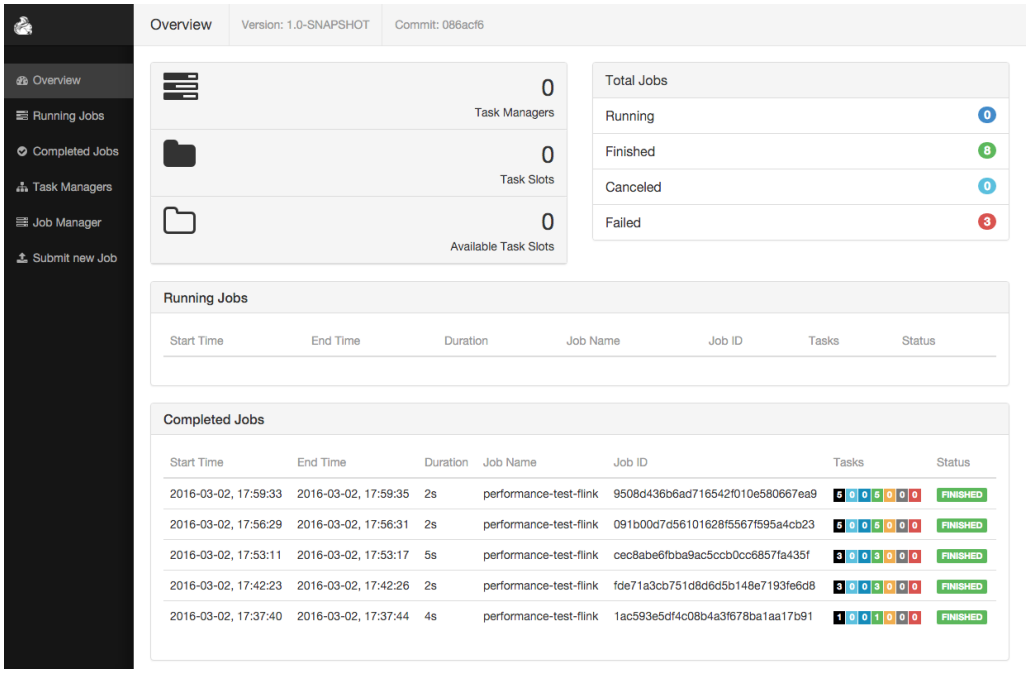


Figure 22: Web dashboard in Flink with pieces of information of the cluster and running and completed jobs.

Flink supports several partitioning strategies. The default is a standard round-robin partitioning over the whole system. When the data is coming from distributed parallel sources, it may occur that one source receives data faster and thus transmits the data to all possible task slots. In order to allow local round-robin partitioning instead of sending data over the network, Flink's API supports the transformation called `dataStream.rescale()`. In contrast to that, `dataStream.rebalance()` partitioning creates equal load for the whole cluster which is especially important for the data skew.

Graphical interface Flink's web dashboard serves similar information about running jobs as Storm and Spark. The landing page, as presented in Figure 22, summarizes most of the important information about the cluster and the jobs:

- the number of registered task managers, all tasks slots and those tasks slots that are still available
- summary of the running, finished, canceled and failed jobs
- more detailed overview of the running jobs
- information about past jobs

After selecting the running or any completed job one can consider the details of that job including the job plan. While a current stream is being processed, the current duration time,

details of sent and received data (in bytes) and records are displayed. For a distributed case it is of great interest to observe the distribution of the events over the different task slots and hence different taskmanagers.

2.5 Summary

All the presented frameworks for distributed Big Data processing are similar in their basic job structure. Despite differences on the API level the data flow through the processing functions is defined as a directed acyclic graph. Storm as one of the first distributed stream processing frameworks introduced a low-level API in constrast to Spark and Flink. The later ones provide basically similar to each other API concepts that are intuitive when building the job graph.

Every framework showcases the job history, an overview of the available resources and especially currently running tasks. This is a significant tool to understand, debug and tune own jobs. Storm, Flink and Spark display all the mentioned details with a lot of information from the general job overview down to every single component. Additionally to that Spark Streaming displays several graphs for the current job and allows for a very detailed insight into each micro-batch. Although Flink's UI is instead held rather flat, it does not loose informational content as Flink does not require as much tuning as Spark. Flink lacks the charts that are comparable to Spark streaming overview. As an advantage over Spark and Storm is that Flink dashboard does not require reloading and updates the views itself. Similar to Spark Flink's dashboard also presents statistics about current memory and CPU consumption of the taskmanagers in the `taskmanager` view (reachable through the left menu).

3 Machine Learning for Distributed Big Data Environments

The motivation behind collecting data is to analyze it and detect patterns or correlations that are not obvious due to the size and complexity of the data. The extracted knowledge is often used to improve a service or a product. Consider a recommendation system that is improved based on previously selected shopping items or a more precise traffic prediction based on the interconnected navigation systems in the cars. Machine Learning (ML) algorithms are developed with the same goal: analyzing and learning from given data and making predictions for unseen data driven by a learned model. With Big Data training a model on a single machine becomes difficult. Although Rapidminer [31], WEKA [32] or scikit-learn increase their attempts to support distributed computations using MapReduce, Spark and other frameworks, this thesis mainly concerned with ML libraries that were built with a clear focus on distributed computing and support for open-source platforms Storm, Flink and Spark.

3.1 Theoretical Background

Depending on the type of the data, different ML algorithms may be required. Two types of ML approaches exist which learn from different types of data. The first case handles labeled data that is given as

$$(X_{1,j}, \ldots, X_{p,j}, Y_j) \quad \forall 1 \leq j \leq n$$

with the number of features p and the number of data instances n. As the classes for the instances are known, **supervised learning** methods can be applied. Such a supervised algorithm trains a model based on the inferred relation between the data vectors $\vec{X}_j$ and the corresponding classes Y_j. If the labels are not known, then **unsupervised learning** methods are able explore the data set and detect multiple classes in it. In such a case only the data vector with no classes is given:

$$(X_{1,j}, \ldots, X_{p,j}) \quad \forall 1 \leq j \leq n$$

Unsupervised algorithms assign each data instance a detected class and do the same for new data.

With a specific algorithm in mind the observed data is used to train a model that assigns each data item a predicted output: $f(\vec{X}_j) = \hat{Y}_j$. As the task is to find a model that fits the data best, an objective function $L : Y \times Y \rightarrow \mathbb{R}^{\geq 0}$ is defined to capture the properties of the learned model such as minimizing the distance between the predicted and the true label. During the training this objective function is minimized or maximized depending on its definition. The algorithm then searches for a function $f(\vec{X}_j)$ that minimizes the loss function:

$$\min \sum_{j=1}^{n} L(Y_j, \hat{Y}_j)$$

Further constraints such as regularizers are often added to the objective function to penalize model complexity. This plays an important role when applying ML algorithms to Big Data. While Big Data may correspond to the number of instances n with $p \ll n$, sometimes the feature space has a very high dimension with $n \ll p$. Therefore, regularization terms are used to perform feature selection.

In both cases the aim is to predict the outcome label Y by applying the trained model on the new data $\vec{X}$. The most frequently used categories of such algorithms are the following:

- for a **classification** (supervised learning) setting two or more labels (e.g. classes, groups, ...) are given and the goal of the classification algorithm is to assign the right label to an unseen data item. Predicting multiple labels for a new data item is also possible (multi-label classification).

- in a **regression** (supervised learning) setting the predicted output $Y \in \mathbb{R}$ is continuous. In the learning phase the algorithm explores the dependencies between the features $\vec{X}$ and the label Y and finds a function that fits the training data best.

- **clustering** (unsupervised learning) devides the input data into clusters. *k-Means* is one of the most known clustering algorithms that partitions the data into k clusters depending on the distance of a data point to the mean coordinates of all the data points in the cluster.

Ensemble Learning With multiple accurate and diverse models it is often possible to achieve more accurate classifiers as in the case when using a single model [33]. This approach is called *ensemble learning*. Various ways to construct an ensemble are known. The two most prominent ones are to split the data *vertically* into subsets of features or *horizontally* into subsets of data instances. The models are then trained using those subsets.

A well-known example is Random Forest [34], where multiple decision trees are trained on different attribute sets of the data. In the end they are combined to a powerful classifier based on the majority vote upon all trees of the forest.

Mining Streaming Data Different techniques of ML introduced above imply that model training is performed in a batch manner. However, as introduced in section 2 data streams continuously produce new data items. As a simple approach the data items from a stream can be stored and used for model fitting with the same batch processing mode. This architecture requires regular reprocessing of the new data that might have an impact on the model performance. Therefore, the strategy suggested above does not fit into the needs of applications where real-time responses with the most current model are necessary. Hence, this makes the development of ML algorithms, capable of mining streaming data without a need to store the data, necessary.

The streaming ML paradigm implies that the corresponding ML model for data mining utilizes real-time feedback and model updates are fast.[22] Therefore multiple algorithms have been adjusted to learn and update their models directly on the data stream. Such algorithms must deal with data whose nature or distribution changes over time [35] and thus the detection of concept change is crucial for non-stationary environments. Current publications present ways of learning various learning tasks on data streams such as multi-label classification [36], adaptive model rules [37] or an ensemble of adaptive model rules [38]. MOA [39] provides several stream clustering, classification and regression algorithms. In this work a distributed version of MOA called SAMOA is introduced next to other distributed ML frameworks in section 3.3.

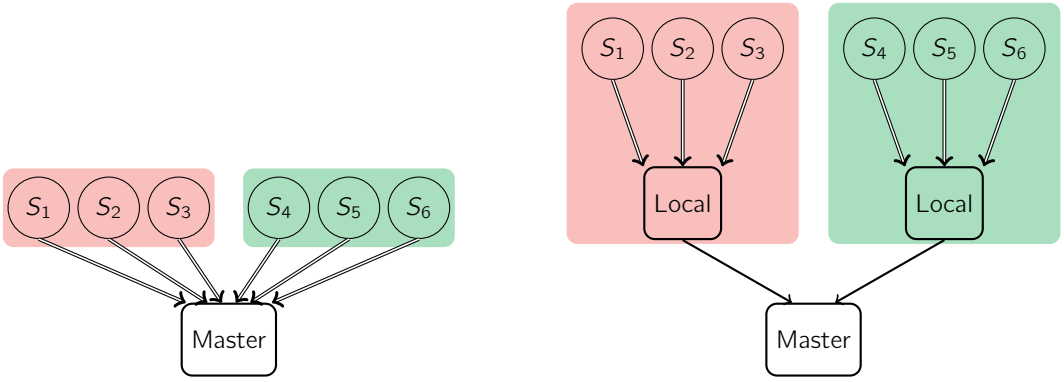
3.2 Machine Learning in the Distributed Setting

The development of multiple frameworks for distributed processing led to an increased research and development of distributed ML algorithms. In order to learn a model one may consider a centralized scheme with a master computing node. This node communicates with all the machines, receives the data and builds the model itself [40]. Due to a variety of reasons like low bandwidth, long response time or load balancing such a scenario becomes less attractive. Hence, new approaches aim at avoiding the collection of all the data on a single central machine. That is, a global problem is split into many small subproblems which can be distributed over a cluster and solved by local algorithms [20, Chapter 14].

Consider a sensor network with the sensors distributed over different locations as shown in Figure 23. In a centralized scheme all the data is transmitted to a computing node that processes it. The transfer costs are high and slow down the performance. When the global problem is split into several subproblems, the data does not have to be implicitly transmitted to the master node. Each subproblem is solved locally where the transfer costs for the nearby located sensors are low and only local models are forwarded to the master to update the global model. This reduces the communication costs and the power consumption while the computation is able to run with a higher level of parallelism [20, Chapter 13].

ML in MapReduce Chu et al. [41] introduced the concept of using ML algorithms inside a MapReduce framework. The mappers divide the global computation into several subproblems. The reducers combine the results and in case of ML generate the trained model. The concept for the distribution of the ML algorithm's computation remains the same for other distributed processing frameworks like Spark oder Flink where the MapReduce scheme is split into more fine-grained functions.

[22] http://www.otnira.com/2013/10/06/samoa/



(a) Centralized ML scheme **(b)** Decentralized ML scheme

Figure 23: Different models for ML in a distributed setting. 23a presents a centralized appoach where every sensor transmits the data to a master node. In 23b the data is only transmitted locally and the solution of the subproblem is transmitted to a master node.

Distributed ML Algorithms To fit the model, an objective function is defined which is computed iteratively over the whole data set. In the distributed setting the data set is not placed at a single location and hence the algorithms need to be adjusted for distributed model training if possible. Many state-of-the-art ML algorithms require only few changes for the distributed execution, some of these are presented in the following.

Naive Bayes is one of the oldest probabilistic classifiers. For a data instance $\vec{x}$ Naive Bayes computes conditional probabilities for each class C_k. Consider the feature vector $\vec{x}$ with all features conditionally independent of other features given the class C_k:

$$P(x_i|x_{i+1},\ldots,x_p,C_k) = P(x_i|C_k)$$

Then the probability for a class C_k is

$$P(C_k|x_1,\ldots,x_p) = \frac{1}{P(\vec{x})}P(C_k)\prod_{i=1}^{p}P(x_i|C_k)$$

where $P(C_k)$ is the number of examples with class C_k out of all examples and $P(x_i|C_k)$ the number of examples with feature x_i labeled with C_k. The classification task is stated as $\arg\max_k P(C_k|x_1,\ldots,x_p)$ whereas the scaling factor $\frac{1}{P(\vec{x})}$ can be ignored. Generally speaking for Naive Bayes one only needs to count the examples given classes and the features. In a distributed environment collecting all the statistics can be split over the computation cluster and the results are then combined into a Naive Bayes model. In section 2.1, frameworks for distributed processing were described. Among other features, they provide support for grouping the incoming data by a key that is required for this setting. The data items are thus grouped by some key (e.g. class label) and counts are computed in parallel on the different machines. Instead of a simple grouping, vertical parallelism as discussed above may improve the load balancing, parallelization and network communication costs. For this each data item

is split among the features and hence an example instance is processed in parallel on multiple machines. Furthermore, the duplication of the computed statistics for the single feature is reduced as it is held on one single machine.

k-Nearest-Neighbor (kNN) is a widely used classification and regression algorithm. Instead of using the whole data set for a prediction, only the nearest k instances are chosen. By considering those instances, a prediction can be made by a majority vote among the k examples. In a distributed environment the most direct method is to split the incoming data into subgroups, search for a neighbor in each of those subgroups and then consider the k neighbors found among the distributed subgroups [42]. In MapReduce structure the pair-wise distance can be computed during the *map* step while the *reduce* step takes over the search for k smallest distances.

k-Means is a clustering algorithm that divides the data into k partitions. The goal is to minimize the distance of all items in each partition to its centroid. A data point is classified to a cluster, if its distance to the centroid of the cluster is minimal among all clusters. In a distributed environment the data is split into subgroups and then the clustering is applied to those subgroups. Forman and Zhang [43] introduced an approach for various distributed clustering algorithms including k-Means. In this case, each worker node computes its contribution to the global statistics and broadcasts them. Afterwards, the centers can be adjusted locally on every machine providing identical results throughout the distributed setting. The algorithm stops when the performance converges or after a predefined number of iterations.

3.3 Distributed Machine Learning Frameworks

With the evolving frameworks for distributed Big Data processing a high amount of different approaches to distribute ML algorithms over several physical machines were developed. In the previous section the general ideas for distributing ML algorithms were discussed. Using three state-of-the-art ML algorithms some ideas on their distributed variants has been shown. In this section several distributed ML frameworks that support a wide range of different ML algorithms for batch and streaming distributed execution are outlined. Here, the focus is set on frameworks that are either decoupled from the underlying distributed processing framework or run on top of the three frameworks discussed in this thesis, namely Storm, Spark or Flink.

3.3.1 FlinkML

FlinkML is a machine learning library provided by Flink. As it is still in development, the support for distributed ML algorithms is rather limited.

The main concept behind FlinkML are ML pipelines, inspired by sklearn.[23] Buitinck [44] conceptually describes pipelining used in sklearn. In the ML context the user creates chains of operations to transform the data (e.g. create features) and then uses the transformed data for a predictor or simply outputs them for any other task. The API of FlinkML[24] is contained in `ml.pipeline`. Like in sklearn the following interfaces are present:

- **Estimator** contains a `fit` method and is the base class for the other two interfaces, *transformer* and *predictor*.

- **Transformer**, as the name states, is used to transform the data and hence has `transform` additionally to `fit` method. As an example one can consider the *standard scaler* transformer which scales all features to the user specified mean and variance. During the training phase the `fit` method learns mean and variance of the data and afterwards the `transform` method alters the values according to the user specified mean and variance values (otherwise, default values of 0 and 1 resp. are used).

- **Predictor** interface makes predictions using the `predict` method and trains the model using the same `fit` method as the other interfaces. Hence, as for transformer the `fit` method is used while training the model and `predict` is called on the test data. In the case of supervised learning, the predictor returns a label for classification or a value for regression, but one can also implement a predictor for unsupervised learning tasks.

A chain of transformations can be implemented by calling `chainTransformer` and is not limited. On the other hand a predictor is added to the pipeline by calling `chainPredictor` and always means the end of the pipeline. Following this definition, a pipeline ending with a transformer can be used as a transformer. If the last estimator is a predictor, then the whole pipeline is used as a predictor. Each estimator needs a predefined list of input and output types and before deploying the job to the cluster, FlinkML inspects the pipeline for type matching, making it type-safe. An example of building a FlinkML pipeline is presented in Figure 24.

The number of algorithms supported by Flink (August 2016) is still rather small, although various algorithms are in development.[25] An overview and comparison is presented in table 4. As already mentioned FlinkML contains some data transformers such as different scalers. Splitting the data set into train and test set or even preparing a k-fold train-test set is possible with a cross-validation `Splitter`. These features are basicly enough to build a ML pipeline.[26]

[23] `http://scikit-learn.org/stable/`

[24] `https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/ml/pipelines.html`

[25] `https://github.com/apache/flink`

[26] `http://de.slideshare.net/tillrohrmann/machine-learning-with-apache-flink`

```
// Training data
val input: DataSet[LabeledVector] = ...
// Test data
val unlabeled: DataSet[Vector] = ...

// Construct the pipeline
val pipeline = StandardScaler()
  .chainTransformer(PolynomialFeatures())
  .chainPredictor(MultipleLinearRegression())

// Train the pipeline (scaler and multiple linear regression)
pipeline.fit(input)

// Calculate predictions for the testing data
val predictions: DataSet[LabeledVector] = pipeline.predict(unlabeled)
```

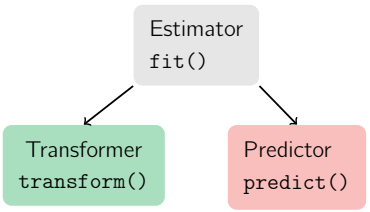


Figure 24: FlinkML pipeline example written in Scala using Multiple Linear Regression

3.3.2 Spark's MLlib

Spark's MLlib introduced more machine learning algorithms and operations for distributed streaming environment. The development of MLlib started as a project called MLBase [45]. MLlib is written in Scala and uses native (C++ based) linear algebra libraries on each node [46].

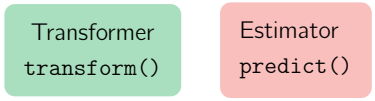


Figure 25: Components of a MLlib pipeline.

As in FlinkML, Spark's MLlib builds ML pipelines as it is used in sklearn [44]. Although the concept is used in a similar way, it is reduced to the two terms as in Figure 25: *Transformer* and *Estimator*. The transformer implements the `transform` method. The estimator is equal to a predictor in FlinkML and implements the `fit` method. The model trained by an estimator is again a transformer that turns a data set with features into a data set with predictions. This way a MLlib pipeline is not restricted to end with an estimator as in FlinkML. Hence, a predictor from FlinkML is a combination of a transformer and an estimator in MLlib: the estimator's `fit` method is used for training and the transformer's `transform` method used for making predictions. Figure 26 shows example code for building a pipeline in MLlib. The task is to tokenize the words, transform them into feature vectors and then learn logistic regression model. As described above after the `fit` method is called on the pipeline in line 14, the received model is again a transformer that can be used for predictions, model validation and model inspection.

As each of the transformers and estimators is controlled by several parameters it requires the right tuning for the given combination of the data and the learning goal. MLlib supports cross-validation for this hyperparameter tuning.

```
1  // Configure an ML pipeline out of three stages:
2  // tokenizer, hashingTF, and lr.
3  Tokenizer tokenizer = new Tokenizer().setInputCol("text")
4          .setOutputCol("words");
5  HashingTF hashingTF = new HashingTF().setNumFeatures(1000)
6          .setInputCol(tokenizer.getOutputCol())
7          .setOutputCol("features");
8  LogisticRegression lr = new LogisticRegression()
9          .setMaxIter(10)
10         .setRegParam(0.01);
11 Pipeline pipeline = new Pipeline()
12         .setStages(new PipelineStage[] {tokenizer, hashingTF, lr});
13 // Fit the pipeline to training documents.
14 PipelineModel model = pipeline.fit(training);
```

Figure 26: MLlib pipeline example written in Java using Logistic Regression

3.3.3 Apache SAMOA

Apache SAMOA stands for Scalable Advanced Massive Online Analysis and provides machine learning algorithms optimized for streaming cases [47]. Its developers have already built the non-distributed stream mining library MOA with a great number of algorithms.

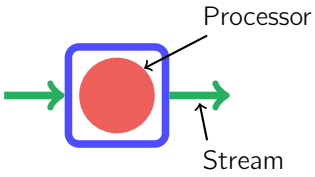
SAMOA is a framework and a library at the same time. It provides an abstraction layer for the distributed streaming ML algorithms such that the developers do not have to focus on the exact processing engine like Flink, Storm or Spark that executes the algorithms. Hence, new distributed streaming ML algorithms can run on different distributed stream processing frameworks.[27] The architecture borrows the terminology from Apache Storm. SAMOA defines a *Topology* that is set up by the *TopologyBuilder*. The following components of a topology are shown in Figure 27a:

- *ProcessingItem* is a hidden component that wraps *Processors*

- *Processor* is a container for the algorithm code that can receive data from one source, but is able to split the output over several destinations. Additionally to a normal processor there exists an *EntranceProcessor* which generates the initial stream. Adjusting the level of parallelism allows for splitting the computations over the predefined number of *ProcessingItems* working in parallel.

- *Stream* connects the parts of the topology. As with Storm, a stream can come from a Spout or from a Bolt. These underlying differences are hidden through the SAMOA API .

[27] http://www.otnira.com/2013/10/06/samoa/



(a) SAMOA's processor is wrapped into a processing item and outputs new stream.

```
// create Stream
Stream stream = builder.createStream(sourceProcessor);

// create destination Processor
destProcessor = new HelloWorldDestinationProcessor();
builder.addProcessor(destProcessor, 4);
builder.connectInputShuffleStream(stream, destProcessor);

// build the topology
helloWorldTopology = builder.build();
```

(b) Building SAMOA topology using its abstract components.

Figure 27: Apache Samoa

- *ContentEvents* are key-value wrappers for the messages transmitted between the ProcessingItems. The key is used for various groupings as in distributed processing frameworks that have been introduced in section 2.4.

Figure 27b shows an example of how one can combine processors to a SAMOA topology. As can be seen, the processors and streams are added to the builder first and are then connected to each other by the API.

SAMOA includes algorithms for distributed stream mining such as for classification, clustering or regression. Furthermore, next to other algorithms SAMOA includes such meta-algorithms as boosting and bagging. The authors suggest that a combination of single-machine classifiers e.g. from MOA with these meta-algorithms is the way to go. Therefore, an additional SAMOA-MOA package is available, while increasing the number of streaming ML algorithms.

Based on the terminology and the idea behind Storm, SAMOA provides a DAG that can be ideally mapped onto Storm or other stream processing engines that use DAG to define the distributed jobs. Hence, SAMOA supports *Storm* and *Flink* (not officially announced, but the implementation is a pending pull request) next to S4 and Samza. As Spark's API is not really different and is also based on the same concepts as Storm and Flink, it is possible to implement the required SAMOA adapter components for Spark integration.

3.3.4 StreamDM

StreamDM [48] is a stream mining library built on top of Apache Spark Streaming at Huawei Noah's Ark Lab.[28] It is one of competitors of the Apache SAMOA framework, while Albert Bifet is one of the developers of MOA, SAMOA and StreamDM frameworks. The main dif-

[28] http://huawei-noah.github.io/streamDM/

ference between SAMOA and StreamDM is that StreamDM is coupled with Spark Streaming platform.

The algorithms provided by StreamDM are presented in the overview in the Table 4. The platform is built with simple extensibility in mind. It has a well and yet simply defined structure for creating new tasks and learners which are presented in the following.

Data structures Beginning on the data level an *instance* in StreamDM holds a multidimensional vector of data. The real data can be saved as an array of double or non-numeric (string) values. Such operations as `dot`, `distanceTo` or state-of-the-art `map` and `reduce` are supported for the instances. Several representations are available such as *DenseInstance*, *SparseInstance* or *TextInstance*.

Example			
input	:	*Instance*	(data, x_{*j})
output	:	*Instance*	(labels, y_j)
weight	:	*Double*	

Table 2: Structure of *Example* in StreamDM

The instances are wrapped inside an *example* object that is passed through DStream. This wrapper holds the input and output instances. The input instance is the real data coming from the source. In the output instance StreamDM saves the labels. Additionally each example can receive a weight. However, output instance and weight are optional. For ease of understanding these data structures are presented in Table 2.

Task Tasks are sequential algorithms that use a Spark Streaming Context. A task is a combination of blocks that are generally ordered as in any learning process setup:

1. *StreamReader (read and parse Example and create a stream)*
2. Learner (provides the train method from an input stream)
3. Model (data structure and set of methods used for Learner)
4. Evaluator (evaluation of predictions)
5. *StreamWriter (output of streams)*

In the beginning predictions are made using the current linear model. Afterwards, using the incoming examples, a new model is trained and updated.

3.3.5 Other Frameworks

Apache Mahout Apache Mahout is a library of scalable machine learning algorithms[29]. Most of the implemented algorithms run on Hadoop's MapReduce, but support for other platforms as Spark, Flink or H2O is growing. In the most current releases Mahout focused on

[29] `http://mahout.apache.org/`

linear algebra, statistical operations and data structures included within the math environment Samsara. As can be seen in the list of available algorithms in Table 4, Mahout currently does not have a big range of algorithms for Spark or Flink. Furthermore, the only streaming algorithm (streaming k-Means) is listed deprecated for the engine version 0.12.0.

As another disadvantage mentioned in various sources [7, 49], most of the available documentation is outdated and the details of implementation can often only be understood by inspecting the code itself.

Therefore, Apache Mahout is mentioned here without further details. Although it was a popular ML tool for Hadoop's MapReduce, it is not as rich in features as other frameworks discussed in this thesis.

H2O H2O is another library for distributed machine learning algorithms developed by the company H2O.ai which sells enterprise support for its open-source library.[30] H2O can be executed both as a standalone application and inside a Spark or Hadoop cluster. It provides a wide range of different algorithms including PCA, k-Means, Random Forest, SGD, Deep Learning and many more. A web-based user interface, Flow, makes creation and analysis of ML pipelines accessible for users without good programming skills.

In addition to its own processing engine, H2O offers seamless integration with Apache Spark and Spark Streaming, through the so-called *Sparkling Water* project. The development continues as H2O announced support for Spark's version 2.0 even before it has been officially released.[31] Storm is also compatible with H2O. Documentation on how to make real-time predictions using H2O on Storm already exists.[32] Integration of Flink is currently in development.

Despite multiple claims of the superior engine's speed and the intensive combination with Spark, it is difficult to find any academic neutral studies verifying these statements and comparing it to other frameworks. Kejela et al. [50] used H2O in order to prove its ability to distribute ML algorithms and scale for Big Data. Unfortunately, although Spark is mentioned, no further comparison in executing time or results have been made. Nevertheless, H2O might become an interesting companion for distributed Big Data analysis with its rich documentation and the intensive support for Spark.

3.4 Summary

Compatibility of distributed streaming frameworks with distributed machine learning libraries is shown in Table 3. Table 4 presents some of algorithms, data processing and utilities implemented in the libraries presented in this section. Spark's native library MLlib provides

[30] http://www.h2o.ai/

[31] http://www.h2o.ai/press-releases/2016-07-01-H2O-Unveils-Sparkling-Water-2/

[32] http://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/streaming/storm/index.html

high number of algorithms implemented for its own distributed framework and at the same time has the best support for various distributed ML frameworks. H2O is definitely as interesting as MLlib as it is the only framework to support Deep Learning out of the box. Thus, Spark's common project with H2O, Sparkling water, becomes attractive where the user can profit from both systems. Flink started to work on their own library much later and has a long list of algorithms to be implemented in the near future. Nevertheless, one can find a lot of discussions and current work on integrating support for Flink into libraries mentioned above. Although Storm has been introduced earlier, at the moment it does not reach the same support for machine learning libraries as Spark.

FlinkML, Spark's MLlib, Samoa and StreamDM are promising candidates for previously introduced distributed stream processing frameworks such as Spark, Flink and Storm. FlinkML, MLlib and StreamDM have been specifically developed for the distributed stream processing and hence fit well and are simple to integrate into existing solutions. SAMOA provides a good documentation and a concept for distributing its computation on other frameworks, which is also the aim of this thesis. Mahout slowly becomes obsolete and the difficulties in documentation make it less attractive. On the other hand, H2O and its subproject Sparkling Water still misses some academic approvements and comparisons, but showcases a wide range of ML algorithms and a tendency of distributing its computations and using its power on top of other frameworks including Flink and Storm. KDNuggets stated 2015 that tools like H2O and MLlib are among distributed ML frameworks with strong growth [51]. At the same time Rapidminer, one of the most popular tools for data mining and data science according to KDNuggets, added some algorithms from H2O such as Deep Learning and GLM and support for distributed computations on Spark and Hadoop.[33]

	Flink	Spark	Storm
Flink ML	x		
MLlib		x	
SAMOA	(x)		x
StreamDM		x	
MAHOUT	(x)	x	
H2O	(x)	x	(x)

Table 3: Compatibility/integration of distributed streaming frameworks with distributed machine learning libraries. *(x)* hints frameworks with not full support or being in development.

[33] `https://rapidminer.com/products/radoop/`

	Flink ML	MLlib	SAMOA	StreamDM	MAHOUT	H2O
Linear SVM	x	x				
Random Forest		x				x
Gradient Boosting		x				x
Gradient-boosted Trees		x				
Bagging				x		
Decision Tree		x				x
Linear Regression	x	x				x
Streaming Linear Regression		x				
Logistic Regression	x	x				x
Isotonic Regression		x				
Multiple Linear Regression	x					
Naive Bayes		x		x	x	x
k-Means		x			x	x
Bisecting k-Means		x				
Streaming k-Means		x				
Streaming k-Means++				x		
CluStream				x		
Gaussian Mixture Model		x				
Power Iteration Clustering		x				
Latent Dirichlet Allocation		x				
Streaming Clustering/k-means		x	x			
Alternating Least Squares	x	x				
Principal Component Analysis		x			x	x
Frequent Pattern Mining		x				
Streaming Frequent Itemset Mining			x			
Deep Learning						x
QR Decomposition					x	
Stochastic SVD		x			x	
Hoeffding Decision Tree			x	x		
Adaptive Model Rules			x			
Stochastic Gradient Descent	x	(x)		x		x

Table 4: List of supported machine learning algorithms in libraries for distributed computing. SAMOA targets streaming (online) machine learning that are limited. MLlib and H2O have support for even more exclusive algorithms. MAHOUT supports more approaches for MapReduce.

4 Metaprogramming with `streams`

Since the introduction of MapReduce, a lot of Big Data processing moved to batch processing on clusters. Furthermore, the definition of such distributed jobs shifted from the two-stages concept of MapReduce to a more granular way in the current distributed processing frameworks. In section 2.4, some of such distributed processing frameworks, namely Storm, Spark and Flink, were introduced. All these different platforms define their jobs as directed acyclic graphs (DAG) with the functions wrapped into the nodes and the data flow interpreted as edges. The differences of the platforms are given by slightly different APIs and the details of the execution runtime environments.

While new projects arise and claim to achieve higher performance, MapReduce is still widely applied for processing huge data loads. On the other side, it is often combined with modern distributed stream processing frameworks in order to build a system following the Lambda architecture concept. Thus, in many usecases the developing process is split into implementing jobs in two different concepts (batch and streaming on top of different platforms). By using for example Spark for batch and streaming processing, the complexity of the development is reduced radically, but all the jobs have to be redefined and adjusted to the new system. Therefore, some projects arised to simplify and improve the job definition for MapReduce and lately other frameworks. That is, some definition language is used to build up processing pipelines that can be parsed and transformed into native pipelines of the different platforms.

In this chapter, first several open-source abstraction tools and the idea behind them are discussed. Afterwards, `streams` framework is introduced that is seen as a standalone streaming engine and an abstraction job definition tool. At the end, it is discussed how `streams` can be used to compete with other abstraction tools to build a Lambda or Kappa Architecture system.

4.1 Open-Source Abstraction Tools

Before the processing of the data is executed on a particular machine in a distributed setting, two more essential steps have to be done. On the top, there is a framework in which the job is composed. The generated task is then executed by the runtime environment of the framework which is responsible for distributing the task over all the computing nodes (cores of the machine or even different physical machines) and monitoring the process. The environment can then be executed on the local machine, in a cluster or in the cloud.

Big Data applications for data analysis require skills in writing code and knowledge in several further areas such as software engineering, testing and statistics. Another challenge is the ever evolving Big Data ecosystem. New MapReduce competitors have been built in the last few years introducing a faster way to process not only batches of data, but also streams

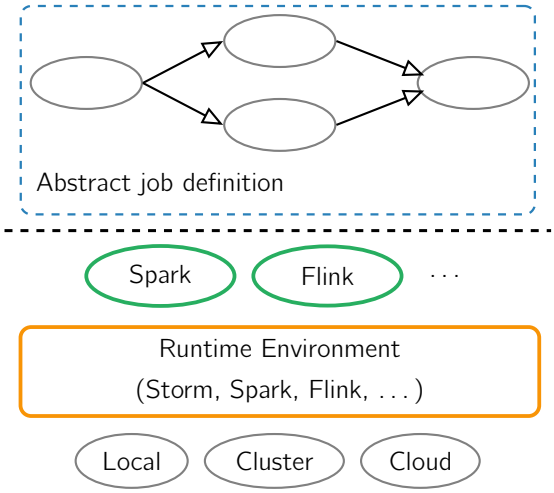


Figure 28: Execution stack of any distributed processing framework. An abstract job definition tool can be put on top of them to decouple the design of the data pipeline from the execution engine.

of data in real-time. Depending on the case and the data itself, some framework may be superior to another one. Instead of implementing each job in a new framework, one way of reducing the programming overhead and hiding the details of the execution environment is to add another abstract layer on the top of the stack described above. As all the discussed frameworks use DAG for job definition, this new abstract layer can use Domain Specific Language (DSL) for the job definition. This enables concentrating on the job definition itself without specifying the execution engine. Figure 28 visualizes this idea. In the following, a short overview over the existing open-source tools is presented.

Tools for application experts One group of such tools is oriented for the data analysts with the limited experience in programming. The syntax for job definition is described by a high-level DSL that is mainly based on SQL syntax.

Apache Hive[34] is built upon Hadoop and allows summarizing, querying and analyzing the data. Hive uses SQL-like language called HiveQL to query and process the data. HiveQL provides the standard SQL functions that can be extended with some user defined functions, aggregates or table functions. As many data warehouses use SQL for the data query, it can be easily performed by Hive while being able to execute complex and intense computations distributedly using MapReduce, Tez or Spark. The code snippet in Figure 29 shows the word-count example in Hive.

A related project, **Apache Pig**, introduces its own new data flow language to define jobs, namely Pig Latin, which is still similar to SQL syntax. Tasks defined in Pig Latin can be executed on top of Hadoop, Tez and Spark. It can be extended by user defined functions

[34] https://hive.apache.org/

```
DROP TABLE IF EXISTS docs;
CREATE TABLE docs (line STRING);
LOAD DATA INPATH 'input_file' OVERWRITE INTO TABLE docs;
CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
 (SELECT explode(split(line, '\s')) AS word FROM docs) temp
GROUP BY word
ORDER BY word;
```

Figure 29: Word-count example using HiveQL

(UDFs), although this leads to the use of a second programming language next to Pig Latin and therefore requires more advanced programming skills.

Apache MRQL (Map Reduce Query Language) was originally intended to produce MapReduce jobs without writing complex MR programs. Although its domain specific language is also based on SQL syntax, it provides support for iterations, streams and grouping as done in modern distributed processing frameworks. The support for Spark and Flink execution engines was added and hence, the same code originally intended for a MapReduce framework can be executed inside of the different frameworks.

Tools for programming experts As a second group, there are frameworks that are more developer oriented as producing data processing pipelines is accomplished by coding against the platform's APIs. In the following, two tools that allow the developers quit thinking in the map-and-reduce concept are illustrated.

Cascading provides a Java API to define data analysis and management jobs. Similar to the concept of Storm's topologies, within Cascading the user combines processing functions called *pipes* into *flows*. For each flow, source and sink taps need to be set which complete the data flow description. Cascading allows translating the defined DAG into MR, Spark or Flink jobs [7].

Apache Crunch similarly to Cascading provides the Java API for the definition of a job. In Crunch, *pipelines* define the data processing flow that can be mapped directly to MR, Spark or local job. With a slightly different API than Cascading, the functions are combined into the pipelines the same way as in any underlying distributed processing framework. After the introduction of lambda functions in Java 8, Crunch has been extended to Crunch Lambda. This new API almost fully matches the APIs of Spark and Flink while it still can be executed as a MR or Spark job [52]. A word-count job written in Crunch Lambda is presented in Figure 30.

The two types of abstraction tools provide a user with an ability of concentrating on the job itself instead of caring about the execution platform and its programming concepts. Abstraction tools such as Hive or Pig are widely spread for data querying and analyzing

```
Pipeline crunch = new MRPipeline(WordCountJobLambda.class);
Lambda.wrap(
  crunch.read (From.textFile("/path/on/hdfs")))
        .flatMap(line -> Arrays.stream(line.split(" ")), strings())
        .count ()
        .map (wc -> wc.first() + ":" + wc.second(), strings())
        .write (To.textFile("/path/to/output"));
crunch.done();
}
```

Figure 30: Word-count example using Crunch Lambda

batches of data [53]. The developer tools as Crunch or Cascading are capable of building complex distributed batch processing jobs. In both cases, it is possible to run the generated jobs on various distributed engines. Finally, combining both types of such tools is possibly the way to go. That is, the developers build the complex processing pipelines using the abstraction tools and the application experts query and analyze the outcoming result data. However, the development is then split over different tools with different DSLs that may still deliver maintenance difficulties.

In the following sections, another approach is introduced that is able to define streaming jobs for the distributed processing frameworks. First, `streams` is described as a standalone streaming engine and a framework. After that, it is showed how `streams` can be used to distribute already defined data processing pipelines onto distributed processing frameworks.

4.2 `streams` Framework

The landscape of various frameworks for streaming and batch applications continues to grow rapidly. Due to fast evolving Big Data technologies the choice of the underlying execution framework is hard and implies high costs for the later exchange of the framework. The `streams` framework [1] is intended to bridge the gap between the abstract modelling of the application flow and various underlying streaming and batch frameworks. `streams` accomplishes the definition of data processing pipelines without any major specification of the execution engine.

The development was driven by several points in mind. At first, there are domain experts with a high knowledge of the application and software architects who construct the underlying processing engine. As mostly these two roles are split, an abstract application design tool need to be as simple as possible to improve the **usability** for application experts. For scientific and productive development the application modelling is an iterative process and hence requires a possibility of **rapid prototyping**. On the other hand there already exist a lot of libraries that solve various tasks from visualizing to machine learning or some specific processing need to be added for the particular case. For such case the framework is easy to **extend** with the

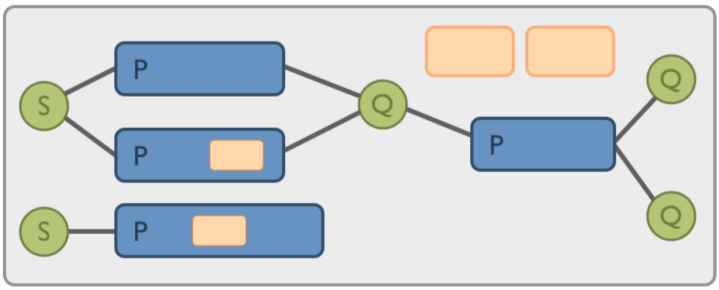


Figure 31: `streams` data flow with stream (S), process (P), queue (Q) and service (orange box) elements [1]

existing library or by UDFs. The definition of the processing pipeline using the high-level API of the underlying framework requires programmatic skills and the precise knowledge of the architecture. All these details are hidden right behind the modeling tool to prevent confusion of domain experts and increase the **platform independency**. For even more complex cases of the systems built following the Lambda architecture, the user should not have to switch to another definition language. Therefore a **multi-paradigm design** allows streaming, batch and service-oriented layers of the Lambda or Kappa architecture to be constructed inside one single framework. `streams` is constructed to deal with these issues.

Data Flow Design The data flow is defined as a directed acyclic graph like in the previously mentioned tools for distributed processing. `streams` utilizes XML based configuration language while preserving simplicity of the data flow definition and enabling rapid prototyping at the same time. As shown in Figure 31, the process flow consists of the following components (tags):

- *stream* defines a source of streamed data
- *process* is a list of processors and executed in a given order. Each data item produced by *stream* is passed through the list of those processors.
- *queue* allows for passing data items from one *process* to another one; some data items are pushed into a queue and set the input of another process to that queue.
- *service* makes an anytime-service available (e.g. some look up service in each of the processors)

All of these components can be accounted as the nodes in the data flow graph. The edges state how the processing functions are interconnected and are defined by the *input* and *id* attributes. The modeled application graph can contain multiple connected components. Hence, each node has ≥ 0 inputs and ≥ 0 outputs. This way `streams` does not add any constraints for building complex processing tasks. A *process* node accepts *input* from a

stream or a *queue* node with an *id* attribute. On the other hand each process can forward items to a queue by defining *output* attribute or using `Emitter` processors. This way the processing data pipeline consists out of *process* nodes which also contain another pipeline of *processors* to be executed.

service nodes are exploited in two different ways. On the one hand they give access to some any-time properties that are requested during the process. Therefore, such services do not participate in the data flow with no input or output edges. At the same time they can be chained into the list of processors. In contrast to other processors such service processors serve as the source of state information and provide an interface to enable sharing it with other processors or any external nodes. As an example two subgraphs are defined with both any-time service nodes and some state-sharing service nodes.

```
<application id="example">
    <stream id="data" class="stream.generator.RandomStream"
                    limit="1000000"/>

    <process input="data" id="separate" copies="2">
        <CreateID />
        <stream.util.AlternateAB key="flag" />
        <stream.demo.Enqueue queues="print"
                    condition="%{data.flag} == 'A'"/>
    </process>

    <queue id="print" />

    <process input="print" id="printoutput">
        <PrintData />
    </process>
</application>
```

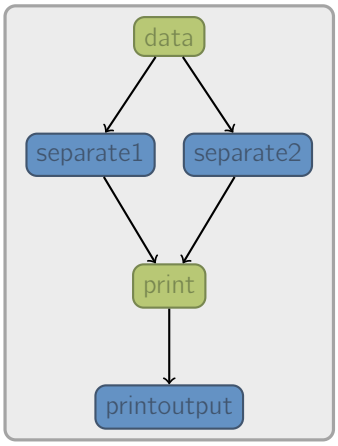


Figure 32: The `streams` application configured by the XML code in the left is visualized as a data flow graph on the right. As the process `separation` uses *copies* attribute, multiple instances are instantiated in order to process the data in parallel.

The code snippet in Figure 32 presents an example of a job configuration in `streams`. One can gather the data flow coming from the stream source, running through one process node and then over the queue to the next process node. The first process node executes multiple processors, one of which enqueues data items matching some condition to a queue. This way only a part of the data stream is reaching the second process node. The queue collects the data from both instances of the first processor and forwards it to the second one.

Data Items The data items passed through `streams` as defined by the `streams.Data` interface are simple `Map` objects. The map data structure is very dynamic as new features or values can be added to the data item during the processing. The keys are `String` values

while the corresponding values have to be `Serializable` objects. This way a data item can be serialized for further message passing.

Extensibility Package `streams-core` contains a list of processors for computing various statistics and manipulating the data items. These processors allow for basic data processing. Nevertheless extensibility is a crucial requirement for a processing framework to enable modeling specific processing applications for different areas. Alongside the XML definition language `streams` provides a simple API for extending its streaming and processing capabilities.

Sources and Sinks The starting point for every application is the **source** of the data. In `streams` every source implements the `Source` interface with the three following methods:

```
void init();
Data read();
void close();
```

These methods describe the life cycle of a source at the same time. After the source has been created, `init()` is called to initialize the source. During the application `read()` method populates the data flow graph with the data items. The source is able to finish its operations such as close the connections to a database within the `close()` method. Another interface `Stream` that extends `Source` interface adds property *limit* to the source that can be set through the attribute in the XML configuration. This functionality comes in handy during the prototyping of the application that can be stopped after reading the set number of data items.

As a second type `streams` introduces a **sink** which receives the data items for further processing. The life cycle of a sink consists of almost the same states. The only difference is that during the application the `write(Data)` method is called. An already mentioned queue implements the `Barrel` interface that combines both a sink and a source because the queue is used as an intermediate actor between two or more process nodes. Hence, some process writes items into the queue whereas the next process awaits data items from this queue to be read.

Processes and Processors The definition of specific processing functionality is as simple as implementing custom sources that was showed above. The **process** element which was mentioned earlier is a container for further pipeline of **processors** that perform the actual processing functions. Yet the process has the same life cycle with `init()` and `finish()` methods for the start and the end respectively. During the running application in the `execute()` method the pipeline of processors is executed in the order given through XML configuration. The data items are retrieved from the input source by calling its `read()` method. All the processed output can be then forwarded to the next process node.

Implementing processing functions to use external libraries or write own data processing logic is one of the aforementioned core concepts. Each processor implements a simple *Processor* interface:

```
public interface Processor {
    public Data process (Data item);
}
```

Adding the code into this method and placing it in the XML configuration is already sufficient for all stateless computations. The parent process nodes takes care of the data items flowing through the ordered list of the processors. In case of any function producing no item (`null`), the process stops execution and continues with the next item.

For every stateful application the framework keeps aware of the state information that is available throughout the whole application lifetime. In order to add such support for stateful processors one has to simply implement `StatefulProcessor` that adds only two methods:

```
public void init (ProcessContext ctx) throws Exception;
public void finish() throws Exception;
```

This way `init` is called before the first data item arrives and `finish` after all the data items have been processed as in other `streams`' components. All the process relevant statistics need to be saved into the `ProcessContext`. The context is a `Map` of not persisted values. Additionally, parameterizing processors happens fully automatically by adding attributes to the processors in the XML definition on the one side and the same-titled variables in the code on the other.

The simplicity of this API makes implementing own processors intuitive. The implementation of stream sources, queues or services is handled in a very similar way. In the course of various projects utilizing `streams` for modeling the streaming applications a number of processors for diverse use cases have been created. All of them can be reused by adding the particular package to the `streams` executable *jar* file and placing the right processor in the XML configuration.

The introduced `streams` framework gives the domain experts the possibility of simple application design without taking the freedom of extending it with some user defined functions. Throughout this whole section no details about the execution engine needed to be mentioned as it stays decoupled from it. In the following section the way of running a `streams` process locally or on top of some distributed processing framework is presented.

4.3 `streams` on Top of the Distributed Frameworks

In the previous section, the core components of `streams` and their extensibility were discussed. Now the execution of a data flow graph using these components can be performed by different engines. The abstract definition of the streaming applications through `streams`

follows the same idea as the modern distributed processing frameworks such as Storm, Flink or Spark. Each process receives the data item, processes it and propagates it to the next functional unit. For the job definition through the XML configuration and in the `streams` API, no knowledge of the execution engine is needed. As `streams` and the modern distributed processing frameworks use data flow graphs for the tasks configuration, `streams` jobs can be executed on top of them. The general procedure is to instantiate *stream*, *process*, *queue* and *service* components of `streams` first and then to encapsulate them into the native functional units of those frameworks. The XML schema definition does not need to be changed and this way `streams` transforms its application to the native topology for the distributed environment.

Especially while prototyping an application it is inevitable to test it before deploying to the production server. For this purpose, `streams-runtime` enables local execution of a `streams` job. Each component is wrapped into a worker thread that performs the execution. The application can be parallelized to take advantage of the multi-core machines. For this the `copies` attribute instructs `streams-runtime` how many instances of a process will be executed such that each of them can receive the data items from the source. The parallel instances read from their stream source and the load is split randomly among the instances. `streams-runtime` enables parallelizing process nodes only.

With the abstraction level of `streams`, the domain experts are able to build the data processing pipelines. The execution engine can simply be chosen or changed to another one afterwards without modificating the processing pipeline definition. At the same time, the multi-paradigm design is achieved automatically as the application can be executed in batch or streaming mode and the services can report the progress to a serving layer.

In the following, two packages for `streams` that make use of the abstract job definition are introduced. First, this concept was used to perform `streams` jobs on top of Apache Storm through the `streams-storm` module. Furthermore, as another example the `streams-moa` module that enables the extension of the existing `streams` processes with streaming machine learning capabilities of MOA is considered. The combination of both packages is used as a proof-of-concept for using further distributed processing engines and distributed machine learning tools.

streams-storm Storm's job graph, *topology*, consists out of spouts and bolts with the data items forwarded within tuples. `streams` job graph needs to be translated into Storm's topology for the execution on top of Storm's engine. Therefore, `streams` components have to be wrapped into the corresponding spouts and bolts. In contrast to `streams`, Storm uses a *push* principle for message passing. That is, the next item from the source is not retrieved through the processing node after it finished applying the function. In Storm the source continuously produces data items.

`streams-storm` package provides the built-in parser that constructs the native topology for Storm's execution engine. Storm uses `Tuple` as the basic unit of data that is basically a *(key,value)* pair. For the message passing across a cluster, these tuples have to be serializable. `streams` definition of a data item can be easily put into the tuple as it is a map of serializable objects. *stream* becomes a *spout* and *process* and *queue*s - *bolt*s. The stream source is wrapped inside a Storm's spout that calls `nextItem` method to retrieve the next data. It is initialized and executed the same way as it is done locally. As the communication is handled by Storm natively, the tuple is then forwarded to the subscriber of this spout. The same way the process node is wrapped inside a Storm's bolt that calls `process` with each incoming data item. The data item is unwrapped from the tuple, processed by the processors pipeline and then wrapped again to be sent to the next bolt. Summarizing, the wrapper for process node follows these steps when its `execute` method is called:

```
execute(tuple)
        => unwrap(tuple)
                => process(Data)
        => wrap(Data)
        => emit(tuple)
```

Its execution achieves the same result as in the non-distributed mode. The *service* tag was not supported for this mapping onto Apache Storm previously. A more detailed description of the implementation concepts is presented in section 5.

streams-moa `streams` was developed in the way that it can include model learners and classificators into the existing data processing pipelines. With the focus on the streaming applications `streams-moa` [54] adopts MOA for the incremental learning tasks inside the `streams` jobs.

MOA (Massive Online Analysis) [55] is a library and a framework for streaming ML approaches. It includes various streaming ML algorithms and can be easily extended with the new ones. As for Storm, one needs to adjust the data items to the native format of MOA and wrap the classifiers into the process elements. A single data item is called an *instance* with the fixed number of features or attributes. For this the *data item* object with the *(key, value)* structure can hold the regular and special attributes with the corresponding values. Every special attribute's name begins with "@" symbol and hence this attribute will not be used for the learning task. The only constraint for the regular attributes is the non-complex Java type such as `String`, `Integer`, `Boolean` and further one.

In contrast to Storm embeddings, MOA learner is wrapped inside of a *processor*. The model is then trained within the execution of `process` method. For this *features* attribute accepts regular expressions to select the features out of all available *(key,value)* pairs in the data item. Additionally, the class label can be set by using the *class* attribute. For the embedded MOA classifier, the `streams-moa` package handles its initialization with the

parameters set using the same XML configuration language. After that, the classifier is used as a normal processor.

The missing part is making the predictions using the trained model. In order to enable this, the processor that trains the model exposes a prediction service with a simple `predict(Data item)` method that returns a serializable response. The prediction is kept inside the "@prediction" key. Now using the predictions one is able to compute the error with respect to the true label for evaluating purposes. With these basic steps `streams` is extendable with ML capabilities.

The usage of `streams-storm` and `streams-moa` has shown the capabilities of `streams` to join multiple modern frameworks inside a simple data flow graph job configuration. Successful modeling of Storm's topologies shows that `streams` can be used as an abstraction layer to process any computation flow which can then be later executed on another platform. As seen above, Spark and Flink also follow the same approach to model their data flow using DAGs. Hence, their processes can be modeled inside of `streams` framework, too. The processors already written for `streams` can be executed inside each of those frameworks. While MOA is not a distributed library, the distributed machine learning frameworks described in section 3.2 run on top of the distributed processing frameworks as they were shown in section 2. Furthermore, it was used as an embedding inside the `streams` framework.

An important point to be discussed is the two levels of granularity defined by the process itself. The processors that are wrapped inside of the process build another data processing pipeline. Such granularity is used to couple several functions into one process that is mapped to a bolt in Apache Storm, the minimal functional unit. This might be beneficial in some cases and gives the application expert an opportunity to balance the computing load itself. This assumes a deeper knowledge of the complexity and computing load of each processor. The modern distributed processing frameworks may balance the computing load themselves even better through their under-the-hood optimizations.

4.4 Lambda and Kappa Architectures using `streams`

As seen above, `streams`' abstraction layer allows for executing configured jobs on top of a distributed processing framework. Although Storm is a framework for distributed stream processing, `streams` can also be used to configure batch jobs that can be distributed. With `streams-mapred` package [56] `streams` was extended with the ability to define and execute MapReduce jobs. Using the combination of `streams-storm` and `streams-mapred`, a system following the Lambda Architecture as suggested by Marz can be constructed. The advantage of this approach is that there is no need to explicitly write new code for Storm and MapReduce. However, there are still some difficulties with this concept. In case of using machine learning to train a model in MapReduce, applying this model on the real-time data

is not straight forward. One would need to export that model into some format like JSON, XML or PMML (predictive model markup language) and than import and apply it inside of Storm (the speed layer). Furthermore, more effort has to be made to combine the results from the two layers.

With Flink and Spark there is no need to pass trained models using another format mentioned above. This task can be accomplished easier as the code for batch and stream processing is almost identical. Defining a data flow graph with `streams` gives us the ability to easily switch between stream and batch processing without writing two different jobs. Once modeling of `streams` jobs into Flink and Spark is implemented, two different pipelines can be submitted: one for fast stream analysis and one to make more intense and slow batch reprocessing. As both, speed and batch layer, are kept in one framework as in the Kappa Architecture, handling and combining the results is simplified. In chapter 3, different machine learning frameworks are mentioned that can be used with Storm, Flink and Spark. Hence, with some additional steps the `streams` abstract design flow can include distributed ML algorithms on top of the distributed processing frameworks.

5 Implementation

`streams` enables rapid prototyping of the streaming applications. As described in the previous section, in a streaming application `streams` consists out of four components:

- *stream*, stream source
- *process*, processing function
- *queue*
- *service*

With this components one can define various streaming jobs in the form of a data flow graph. The nodes are stream sources and processing functions. The edges represent the flow of the data items through the graph. The queues are the hidden structure behind the edges that connect various function nodes and store the data items until they can be processed. The services do not impact the data flow. For many distributed processing frameworks these components can be mapped onto their native ones. While the names might differ, the topologies in Storm, Flink or Spark follow the same scheme and thus `streams` fits into the role of an abstraction layer to model the jobs for them. Many concepts that have been developed for the `streams-storm` package are reused for the implementations of `streams-flink`[35] and `streams-spark`[36].

The aim of this work is to enable the definition of native Spark, Flink and Storm jobs using `streams`' XML job configuration. While the XML structure should be kept the same, some tags and parameters may be added and some ideas of evolving the internal structure of `streams` arised during the implementation for this thesis. In this chapter it is shown how a `streams` job can be mapped to a job in a distributed processing framework, its implementation in general and of the important features introduced in section 3. Hence, at first the focus lies on finding a way to map the basic components of `streams` onto the distributed processing frameworks. After that the features of the distributed frameworks that are essential for many modern data stream applications are discussed. Not all of these features are part of `streams`, but the XML configuration can be extended to support them.

Scheme to Build and Distribute a Job

The procedure for building and distributing a job is identical for all of the distributed computing frameworks mentioned in this thesis. The following steps are performed:

- *construct* the execution plan

[35] `https://github.com/alexeyegorov/streams-spark`
[36] `https://github.com/alexeyegorov/streams-flink`

- *package and serialize* the job plan and all the user defined tasks; everything that can not be serialized is lost
- *distribute* the packaged file to the workers
- *deserialize* and initialize the distributed tasks
- *run* the execution plan

During the construction phase the XML configuration is parsed and all the `streams` components that are discussed in details below are initialized. For Storm, Spark and Flink this is performed using their corresponding *main* classes `storm.deploy`, `spark.deploy_on_spark` and `flink.deploy_on_flink`.

5.1 Data Representation and Serialization

Transferring data between the instances of a distributed system is a core application. `streams` provides a very dynamic and untyped structure for data items. `streams.Data` is a classical map that uses `String` as a key and any `Serializable` object as a value. Without the type safety the application can not verify the correctness of the interconnected processes, but passes the responsibility to the user allowing for a more dynamic job definition.

Next to passing data items along the job graph after building a job topology on the master node, the task itself is distributed to the worker nodes. On the receiver side the job plan need to be reconstructed and executed. For these purposes of transferring the data and the tasks across the different working machines distributed systems use *serialization*. That is, the data structures and the objects are written down to a predefined format which the receiver understands and reconstructs those items. The prepared task with all the resources and implementations is serialized before transmitting it to the worker units. The worker machine deserializes this package after receiving it and is prepared to process the data stream. As the format impacts size and time of serialization, choosing between a Java's built-in and other serializers can change the performance of the system.

An object is serializable if it has a primitive data type or it implements `Serializable` interface in Java. The primitive data types are such as `Integer`, `Double`, `String` etc. All other more complex types have to implement the `java.io.Serializable` interface to enable the persistence. Such complex types contain only serializable fields and at most fields marked with `transient` keyword. When the object is written down, transient fields will not be serialized and hence, their values will be lost. For every object of a subtype `Serializable` the method `readResolve()` is called, when this object is deserialized. This allows for customized initialization be the user.

Serialization is handled the same way on Storm, Spark and Flink. The information that is prepared or calculated on the master machine while building an execution plan is stored inside

```
public abstract class StreamsObject {
    protected abstract void init() throws Exception;
    public Object readResolve() throws Exception {
        init();
        return this;
    }
}
```

Figure 33: Interface for the distributed streaming functions. When the function is to be deserialized, `readResolve()` method calls `init()`.

of serializable objects. In `streams` for this purpose each process holds a `ProcessContext` that is again a simple `Map` with only serializable objects. It is serialized and distributed over all workers together with the task. On the other hand, the processors in `streams` may contain not serializable fields and thus, they need to be initialized on the worker node directly. Therefore, an abstract class `StreamsObject` is defined like shown in Figure 33 which has to be used by all the wrappers for the functions in `streams` (processor, process). This can be used to call `init()` method and initialize all not serializable fields.

5.2 Stream Source

The source of a stream is the component that is capable of producing a possibly unbounded sequence of data items. The real source of the incoming data varies from a messaging queue such as Apache Kafka[37] or ZeroMQ[38] over the distributed storage like HDFS to a simple data file. DSL of `streams` allows for basically two possibilities:

1. native stream source
2. stream source from the `streams` framework

In order to use the natively supported stream sources the user provides the required parameters in the XML configuration and `streams-{storm, spark, flink}` initializes the stream source. Kafka as a distributed streaming platform allows out-of-the-box parallelization of stream source instances which is not originally supported by `streams`. Nevertheless, in this thesis the focus lies on the `streams` sources and the support for the native stream sources is part of the future work. The *stream* interface provides the capability to retrieve the next data item to be processed independently from the particular stream source. In the distributed framework a *stream* object is encapsulated inside the native *data stream* abstraction which simply retrieves the next data item from the *stream*. A possible way of parallelizing of *stream* instances is presented in the following.

[37] `https://kafka.apache.org/`
[38] `http://zeromq.org/`

Parallelism The motivation behind the distributed processing frameworks is to split the amount of the computations and the data over multiple physical machines. A single data stream source becomes a bottleneck unless the stream source is parallelized. In `streams` it is possible to define multiple `<stream ...>` tags and increase the level of parallelism this way. The disadvantages of this method are the low level of flexibility when tuning the number of parallel sources and the uncertainty about all the sources running on the different working nodes.

The distributed environments support parallel sources and allow better job performance. For this purpose while building the execution plan each source has to be advised about where or how to retrieve the data items. Otherwise parallel sources will produce duplicates. In the case of a bounded source like a (distributed) data file the parallel sources split it up into equal parts and process each of it respectively. Thus, the interface `DistributedStream` was introduced to develop parallel sources for `streams`:

```
public interface DistributedStream extends Stream{
        void handleParallelism(int instanceNumber, int copiesNumber);
}
```

The handler for the parallel data stream source initiates all the multiple sources and passes the assigned source instance number and the overall number of parallel sources through the call of `handleParallelism` method. With that interface in mind a stream source `IterateFilesStream` has been developed for the `streams-spark` package to allow for reading multiple files from a storage in parallel. That is, all the files that are found under the path from the *url* attribute and optionally match with the *ending* attribute are split among the parallel *stream* instances. The way to read those files is defined by the another encapsulated *stream* element:

```
<stream id="fact" class="stream.IterateFilesStream" url="/home/egorov/data"
                ending=".csv" copies="2" limit="20000">
        <stream id="_" class="stream.io.CsvStream"/>
</stream>
```

Storm & Flink

Storm's and Flink's API support implementing distributed parallel sources. It automatically tries to distribute them over separated taskmanagers to improve the performance. Simple definition of multiple sources inside of `streams`' XML file would not have the same effect. In Storm the spout implements the method `nextTuple` that is called by the system to retrieve the next data item. The source in Flink uses `run` method that is called once and should run as long as the data stream source can or should retrieve the data items. Both frameworks give the user the possibility of setting the level of parallelism by an additional parameter in Storm or by a `setParallelism` method call in Flink.

```
ArrayList<SparkSourceStream> functions = sourceHandler.getFunction();

List<JavaDStream<Data>> dataStreams = new ArrayList<>(functions.size());
while (functions.size() > 0) {
        dataStreams.add(jsc.receiverStream(functions.remove(0)));
}

JavaDStream<Data> unifiedStream;
if (dataStreams.size() > 1) {
        unifiedStream = jsc.union(dataStreams.get(0),
                        dataStreams.subList(1, dataStreams.size()));
} else if (dataStreams.size() == 1) {
        unifiedStream = dataStreams.remove(0);
}
```

Figure 34: The parallel source in Spark is built as a union of multiple source instances.

Spark

Generally Spark works the same way as Storm and Flink. It provides the method `onStart` for a data stream source that it meant to run like in Flink as long as the data keeps coming in. The difference is though in the concept of increasing the level of parallelism. In order to achieve higher throughput in the Spark Streaming cluster, the user is suggested to create multiple input DStreams and then build a union over them. Therefore, a `streams`' source is wrapped and if this source is able to be executed in parallel, multiple instances of this stream source are created. The topology builder class receives this list and if there are multiple sources, it builds a union over them. Figure 34 illustrates the process of unifying multiple data streams.

5.3 Processing Function

After retrieving the data item from the source it is passed to the processing node that applies a user-defined function on it. In `streams` the smallest function to be applied is a *processor* where many of them are enqueued into a list of processors or a process. For the sake of simplicity the *process* tag is considered as one complex function to be applied. The motivation behind this two levels of granularity is to split complex analysis and processing pipelines into multiple lighter weight functions, *processors*. At the same time with *process* those functions can be ordered and combined differently such that communication costs are also reduced as a process is always executed in a single JVM.

The *process* interface provides the capability of applying the arbitrary function to the next data item. The processing steps of a *process* tag are encapsulated inside a native function of a distributed processing framework. The native function uses the *process()* API to trigger

the processing steps. Figure 35 shows the idea of wrapping the processes or processors inside the native functions of the distributed processing frameworks such as `flatMap`.

The content of the wrapper method is based on the same idea and consists of the following steps:

1. call `process` on the data item
2. collect items written to the queues

In the following more details are given on each framework, but nevertheless this procedure stays the same.

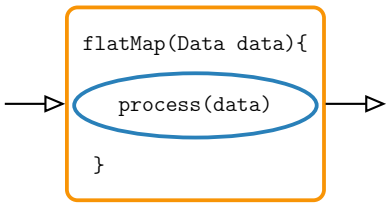


Figure 35: Wrapped `streams`' process function

Parallelism Similar to *stream* each function can be parallelized. The general concept is to run multiple instances of the function inside different threads. The input of the function can come from both the output of another function or the data stream source. The data flow represented by the edges is split among the multiple instances of the function. In the simplest case the data items are shuffled and hence the data load is split equally among all instances. More details on different grouping strategies is discussed in section 5.6.

Storm

In section 2.4.1 bolts have been introduced as the nodes of the data flow graph in Storm. For the use with `streams` the processes are wrapped inside of bolts. Storm's API provides each bolt with multiple methods:

- `prepare`: the equivalent for the `init` method
- `execute`: this method is called for each data item to be processed
- `declareOutputFields`: as Storm works with *(key, value)* tuples the output of the bolt has to define the output tuple for itself
- `cleanUp`: at the end of the processing this method is called to be able to clean up and finish the calculations

Hence, after the deserialization on the worker nodes `prepare` method is called and allows initialization of the process to be finished. The bolt uses `OutputCollector` inside of the `execute` method to pass the data items to the next process. The output written to the queues inside of the process is inspected and forwarded to the next bolt using the collector instance.

Spark & Flink

As discussed before each function extends `StreamsObject` such that `init()` is called again after the function is deserializes on the worker node. Then the function is defined and can apply the list of processors to the data item. In Spark this looks as following:

```
public class SparkProcessList extends StreamsObject implements FlatMapFunction<Data, Data> {
        public void init() throws Exception {...}
        public Iterable<Data> call(Data data) throws Exception {....}
}
```

For Flink's environment the same strategy is done almost identically:

```
public class FlinkProcessList extends StreamsObject implements FlatMapFunction<Data, Data> {
        public void init() throws Exception { ... }
        public void flatMap(Data data, Collector<Data> collector) throws Exception { ... }
}
```

In both frameworks `flatMap` is used as the native function to execute the process and collect the results. `map` function requires one single element as the output, while inside of the process multiple processors may enqueue items to the queues and hence, the output varies from zero to more than one data item. Afterwards the function is retrieved and applied on the data stream.

```
JavaDStream<Data> dataJavaDStream = source.flatMap(function);
```

and the same in Flink:

```
DataStream<Data> dataStream = source.flatMap(function);
```

5.4 Queues

The message passing between the nodes of the data flow graph provides the communication between the functions of a streaming application. Each data item passed from one process to the next one is held inside a queue. The queue is itself a concept of an intermediate actor between the two processes that allows for writing data items into it and retrieving them for the next function. Various queues can be used while `BlockingQueue` is applied as default queue implementation.

In *streams* there are multiple ways of passing a message to the next process:

- `output` attribute in the process
- `Enqueue` processor inside the process that filters data items based on a given condition into a queue
- `Emitter` processor propagates the whole data stream to the queue and hence to the next process

For all the following queue implementations, when the XML configuration file is parsed, the intermediate queues for message passing are collected and are held in a list. The data

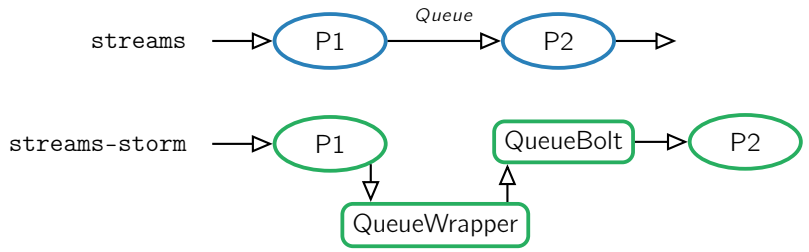


Figure 36: The mapping of a `streams` topology into `Storm` replaces the queue between the processes with `QueueWrapper` to which the processors write the data items and `QueueBolt` that is receiving the data items and forwards it to the next process.

items that are enqueued into a queue should be collected or pushed further through the pipeline. In order to fill the queue with the data items, each emitting processor or process implements the method `setSink(...)`. This way, the emitter (process, processor) and the sink (queue) are interconnected.

In order to provide queue functionality, the `process` is inspected for a *Enqueue* processor in its list and for an `output` attribute. The queue wrapper class adds a label to the data item that should be pushed into the queue. Later in the data pipeline, if the process contains queues, the data items are filtered into different new streams. Each stream is filled with the differently labeled data items. Every framework has its own implementation of it with minor differences.

Storm

Storm uses its `OutputCollector` to directly forward the data items. Hence, the queue itself is not storing the messages. For each process bolt, the queue injection is done. That is, each processor that implements `Sink` interface needs a sink set up which is a queue. Hence, the `QueueWrapper` object is used as a sink that is initiated with the right collector and pushes all the data items directly back into the Storm's topology flow. It is then flowing to the real bolt, called `QueueBolt`, which also does not store the items, but transmits them directly to the collector. This strategy was implemented before this thesis and is represented in Figure 36.

Spark & Flink

The queue implementation adds to each data item a *(key, value)* pair containing the name of the queue (that is, the edge between the processes and hence the data flow). This way, the first process gets the data items from a real source that runs in the background. If this process forwards the data items to the next process, then they are written into the queues. Therefore, all such data items are extended with an entry of the queue name and pushed

forward to the next function. At this point, all the enqueued data items are still in the same data stream. The split itself is done rather similar for Flink and Spark.

Using Flink's API, `DataStream` can be split into several substreams. These new generated substreams are added to the list of the sources that are used as the input for the following processes. For Spark, there are only minor differences to Flink. In contrast to Flink, in order to split the data stream, it is filtered using the queue names. Each of the new filtered data streams becomes a possible source for the following processes like in Flink.

5.5 Services

In section 4.2 two types of any-time services were introduced. Any processor can provide a service and it is automatically initialized together with the processors itself. On the other hand, `streams` allows definition of `service` tags to allow for any-time services that do not participate in the data flow themselves. These services are handled almost the same way as queues and are implemented the same way for Storm, Spark and Flink. During the initialization on the master node the services are initialized and collected into a list. As the services are annotated with a special *service* tag in the code, it can be easily detected. Then during the initialization of the processors the field that requires a particular service can be set.

5.6 Grouping the Data Stream

Each data item of a data stream flows through the job graph. As the main key is to parallelize the processing of the stream, the processors are distributed and hence the question arises how to distribute the data items among the parallel instances of each processor. The very simple strategy is to shuffle the data items, that is, the data load is split randomly across all the processors. The contrary extreme is to propagate each data item to each processor. While this way there is no benefit in parallelization except that different processing is done by the two processors in parallel on the whole data stream. The mostly used grouping is splitting the data items by the keys. In case of word counting one uses the words as the key. Thus, the data stream is split into group of words and each processor or function responsible for counting the words receives the subset of the data stream divided by the keys over all the parallel instances. Figure 37 illustrates this.

Grouping the data items by keys is not supported in `streams` originally, but the XML configuration can be extended to support it inside of the distributed processing frameworks. It can be done at every point of the data flow graph where the data item is forwarded to the next process:

1. *<enqueue.../>*: if a data item is put into another queue to be used for the next process list the user might define a key for different groupings

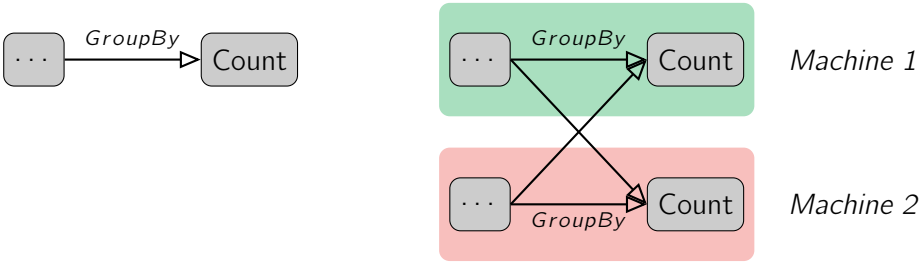


Figure 37: Grouping items of a stream on a single machine (left) and in a distributed setting (right)

2. *<process.../>*: while the process can use *output* attribute to push the data items to the next process, without enqueueing the items one may still want to group them nevertheless; thus defining the key for a process list fits in several scenarios such as for the stateful processing feature in section 5.7.1;

3. *<{customProcessor}.../>*: if each processor is be mapped to a function of a distributed processing framework, then grouping can be performed directly for each function.

The parameter to be used as a key is mostly a *String* or an *Integer* with reoccurring values. Distributed processing framework allow for defining the grouping for the functions. That is, e.g. a *real* number or more complex data structures can be used for grouping.

Storm

Storm has a low-level API. The spout/bolt A that transmits the tuple has to declare the new output field with the given key. On the other end the bolt B subscribing to the spout/bolt A can group with `fieldGrouping` using that key. Also other groupings such as `shuffleGrouping`, `allGrouping` (replicate all items to all bolts) or `globalGrouping` (forward all items to one single bolt) are available in Storm.

Flink

As in Flink there are just minor differences between the `DataSet` and `DataStream` API, one is the grouping API which is `groupBy(''key'')` and `keyBy(''key'')` respectively. The resulting output of the `keyBy` operation is a `KeyedStream`. Flink accepts tuples with more than 2 entries and allows the user to choose the key entry like `keyBy(1)`, `keyBy(0)` or `keyBy("key")`. Even if there were no key entry generated before, it is possible to group the data stream by using some custom function that extracts the key information from the data item as presented in Figure 38.

Storm

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
}

builder.setBolt("counter", new TestWordCounter(true), 4);
         .fieldsGrouping("split", new Fields("word"));
```

Spark

```
JavaPairDStream<String, Iterable<Data>> dataStream = streamSource
        .mapToPair(data -> new Tuple2<>((String) data.get("key"), data))
        .groupByKey();
```

Flink

```
KeyedStream<Data, String> dataStream = streamSource
        .keyBy(data -> (String) data.get("key"));
```

Figure 38: Spark is able of grouping the data stream by *groupByKey*. Flink simply uses *keyBy* for the same task on streams. Storm wants a bolt to declare an output field that can be later used for field grouping.

Spark

Spark and Spark Streaming APIs do not differ at this point. For both Spark and Spark Streaming `groupByKey` and `reduceByKey` methods are available. Although they can only be applied on a `PairDStream`, the standard `DStream` can easily be transformed to this (key, value) stream by calling `mapToPair` to be able to extract some key from the data item. After that the stream can be grouped with `groupByKey` operation. Such an example is shown in Figure 38.

5.7 Further Concepts

The concepts and functions discussed so far ensure mapping `streams` on to distributed processing frameworks. In the following several other features of the distributed engines are discussed. While some are introduced as concepts, others contain details on implementation.

5.7.1 Stateful Processing

Every stream application processes the data either in stateless or in stateful mode. A stateless application provides a response to the received data item based only on the item itself. There is no need to store any information or statistics. Nevertheless, in various applications it is essential to keep the state of the processed data items. That is, one might be interested in the distribution of the words in the stream. Within `streams` a state is called a context.

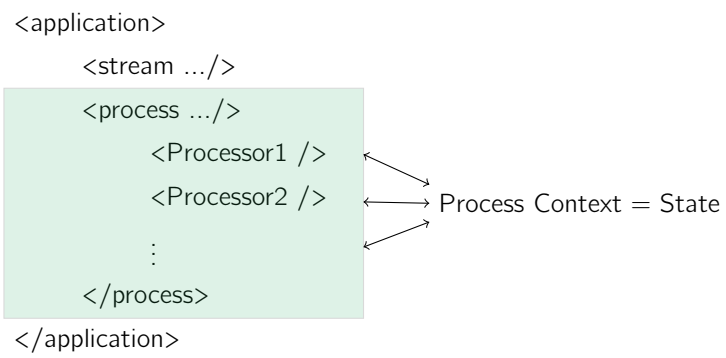


Figure 39: `streams` supports stateful processing by saving information inside a process context.

Although each stateful processor can store some data in the context, many processors used static variables to save and share their intermediate results between the several instances of a process. This works well as `streams` runs inside the same JVM and just creates multiple copies of the running process to make use of the multi-core CPUs.

With the mapping of `streams` processes into the different distributed processing frameworks, it is possible to distribute the processes over several physical machines that logically do not share one JVM. Figure 37 displays such a setup for a word-counting example. Thus, if our intention is to count words in a distributed manner, the context need to be distributed and update. As shown in Figure 39, the process context holds the state of all processors in the given process. Each processor can read information from the context and update or add new pieces of information to it.

Spark, Flink and Storm have support for distributed stateful processing of the data streams. Spark and Flink both have in common that the state can only be saved and updated for streams grouped by some key.

Flink A stateful function in Flink extends *RichFlatMapFunction* and uses a transient variable of type *ValueState* to store the state.[39] Then this function can save the state for the *KeyedStream* (use *keyBy(...)* before the stateful function). Figure 40 displays these two steps: the upper part builds the KeyedStream, the lower code snippet shows the exemplary implementation of retrieving and saving the process context.

Spark The Spark environment provides a special API function called *updateStateByKey*.[40] The user defines the way of how the update step should be done and passes it as a function. Figure 41 displays how Spark can be used to perform stateful processing by using `streams`.

[39] `https://ci.apache.org/projects/flink/flink-docs-master/dev/state.html`

[40] `http://spark.apache.org/docs/latest/streaming-programming-guide.html#updatestatebykey-operation`

Figure 40: Stateful processing can be applied on the keyed data stream in Flink. The state itself can be saved and distributed directly in the implementation of the function (here, *flatMap*).

```
dataJavaDStream
        .mapToPair((PairFunction<Data, String, Data>) data
                -> new Tuple2<>((String) data.get("key"), data))
        .updateStateByKey((Function2<List<Data>, Optional< SparkContext>,
                Optional<SparkContext>>) (v1, v2) -> {
            SparkContext sparkContext = v2.get();
            ...
            return Optional.of(sparkContext);
        });
```

Figure 41: Spark enables stateful processing as Flink on the keyed data stream. The state is updated through applying *updateStateByKey* on that stream.

The first step is to extract a key from the data. Using the key grouping, the state can be retrieved and saved during the second step.

Storm Storm added support for stateful processing in version 2.0.0-SNAPSHOT. A stateful bolt extends `IStatefulBolt` with a single state implementation so far, `KeyValueState` that is a key-value state. As `streams` only stores the context, it is possible to implement a custom state that retrieves and saves the context under the same key. Figure 42 presents the lines of code that allows storing and distributing the context of a bolt.

In the end, all three frameworks use the same scheme:

1. save the state data in the key-value data structure or it is saved on the keyed data stream;

```
public class ProcessStatefulBolt extends ProcessBolt
                implements IStatefulBolt<KeyValueState<String, BoltContext>> {
        private KeyValueState<String, BoltContext> state;

        public void initState(KeyValueState<String, BoltContext> state) {
                this.state = state;
        }

        public void execute(Tuple tuple) {
                BoltContext context = state.get("context");
                super.execute(input);
                state.put("context", context);
        }
}
```

Figure 42: Storm supports stateful bolts since version 2.0.0.

2. before processing the new data, retrieve the distributed state and use it for the current processing
3. store the new context

For the usage with `streams`, there are two main concerns. At first, to be able to save the state each data item has to be propagated through the data flow graph with a key. Three locations for the placement of the "key" parameter have been suggested in section 5.6. As the basic option, grouping the process list by a key is sufficient to enable saving the state of this process. While still using the whole list of processors as a node in the data flow graph, the state is kept only for the process and not for the single processors in the list. That means that in case of a failure the whole list of the processors has to be recomputed. As the second point, the *ProcessContext* holds information concerning the list of the processors. The existing processors may store the data in static variables and hence, such processors will not work properly on top of distributed processing platforms.

5.7.2 Fault Tolerance

As node or machine failure is possible, fault tolerance of a distributed processing framework are one of the essential requirements. That is, the application is able to recover from failures and ensures a specific level of message delivery that was introduced in section 2.3. Operator state distinguishes the two forms of state that the system has to take into account:

- state of a processing function
- system state that refers to data buffers

The first form has already been discussed in section 5.7.1. The state of the system is saved by drawing checkpoints periodically. For this the metadata or the data itself need to be

saved to a reliable file system such as HDFS. Storm, Spark and Flink support checkpointing in the current versions. Checkpointing and restoring system state deals with reliable file systems, configurations of the frameworks themselves and reliable stream sources. Although fault tolerance is an important issue when building distributed processing framework, for this thesis the focus was set to enable a novel abstraction layer to simplify job definition. Improving reliability of such systems is a topic for further future work. Nevertheless, in the following concepts for fault tolerance in Flink and Spark are mentioned briefly whereas a little bit more details are discussed about Storm because of some issues discovered during the evaluation phase.

Spark Spark with activated checkpointing feature allows for retrieving the streaming context from checkpoint data or create the new one otherwise. The concept of RDDs helps to recover from failures as they are immutable and the partitions can be recomputed because Spark remembers the lineage of operations. Hence, only the original RDD need to be saved and this is possible if the batch job is executing on a reliable file system. This is not the case for the streaming applications. Spark provides the ability to replicate RDDs over several executors.

Flink Similar to Spark, Flink draws checkpoints or snapshots of data streaming application periodically. In a case of a failure, the latest checkpoint is used to restore the state of the application. Of course, if the source is not reliable and does not support rewinding the stream to some point in its latest history, then Flink can not guarantee exactly-once message delivery.

Storm The Nimbus and Supervisor daemons in Storm are designed to be stateless and fail-fast. If one of the workers dies, Storm automatically restarts it.[41] Support for stateful processing and checkpointing was announced as part of current developing version of Storm. Unless there is a single stateful bolt, no checkpoint is performed [57].

5.7.3 Windowing

Considering the data stream from a sensor, the time series analysis becomes essential for many applications. Monitoring the product pipeline in the manufacturing sector using various sensors allows for detecting anomalies and changes in the processing pipeline. The process that reacts to such real-time events can improve e.g. the quality of a product or reduce machine failure [40]. As time series data changes over time and approaches continuously for an undefined time period, instead of working on the raw data stream directly, the data items can be grouped into time intervals. In a streaming application, this is called *windowing*. The

[41] `http://storm.apache.org/about/fault-tolerant.html`

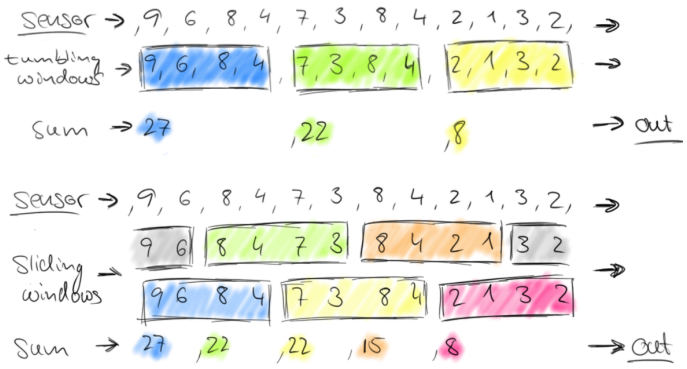


Figure 43: Windows for a data streams can be built as tumbling windows (upper image) or as overlapping sliding windows (lower image).[42]

definition of a window depends on its starting point and the length. Two types of windowing are possible as shown in Figure 43:

- **Tumbling window** collects each data item only once. That is, the second window starting point is set to the next item after the first window: $start_2 = end_1 + 1$. For this type of windowing only the value of the window length has to be set.
- **Sliding window** in contrast to a tumbling window allows overlapping windows. That is, the starting point of the following window is between the starting point and the end of the first one: $start_1 < start_2 \leq end_1$. For a sliding window additionally to the value of the window length one have to define the sliding value.

All the three distributed processing frameworks considered in this thesis provide the native support for windowing. `streams` has a built-in `collect` processor and `foreach` processor list. Using `collect` allows for collecting the incoming data items until a specified number of them has arrived. This is a similar approach to windowing that is bounded by a single JVM. With the `foreach` processor list multiple processors (functions) are applied to each data item from a list of data items.

Combining the native support of distributed processing frameworks for windowing with the `streams` processors makes the definition of windows on `streams` possible. Figure 44 shows this idea. The orange box is the wrapper object for the functions that were discussed in section 5.3. The blue box contains the `foreach` processor list and it receives the windowed data items from the native function. As a subclass of `ProcessorList` the processor `foreach` is a stateful processor and can use its context to share the computed results on each data item. Applying windowing creates a windowed stream that can be transformed back to the original data stream by applying a window or reduce function. In this case `foreach` is used as a reduce function that processes the list of data items and produces one single new data

[42] `https://flink.apache.org/news/2015/12/04/Introducing-windows.html`

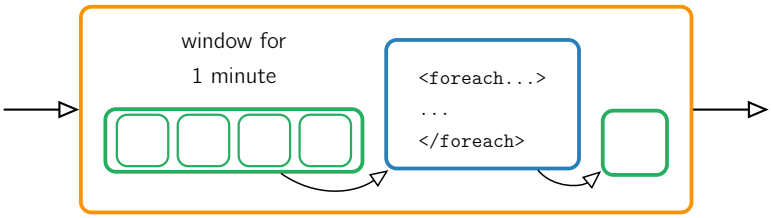


Figure 44: Distributed windowing can combine the existing native windowing functions of the distributed processing frameworks and such `streams` processor as `foreach`. This enables using the same processors from `streams` while computing the results on the windowed stream distributedly.

item containing also the output of all single data items processed. Hence, it is dependent on the implementation of this functions executed inside of the `foreach` processor.

The current mapping of `streams` to Storm, Spark and Flink uses `<process .../>` tags as the function nodes for the data flow graph. One possible solution is to introduce a `<window .../>` tag similar to `<process .../>` tag. Inside the window tag one may directly place the processors that should be applied to each data item of the window. Depending on the execution engine, internally the `collect` processor or a native windowing function of the distributed processing framework forwards the collected items to the `foreach` processor list. The output of the `foreach` process is propagated to the next process by using the same queue concept. The downside in this scenario is that this process does not fit the current `streams` configuration However, `streams` can easily be extended to just transform the `window` tag to the combination of `collect` and `foreach` tags.

5.7.4 Union

In a streaming application sometimes multiple sources are used as input or a single stream is split into several substreams for various parallel processing steps. Depending on the application task after the execution of several functions the user may be interested in unifying multiple data streams into a single one. The idea is rather simple: in the data flow graph a function node f_i subscribes to the data streams from f_{i-1} and f_{i-2} and its output is a single data stream combining the input data streams as presented in Figure 45.

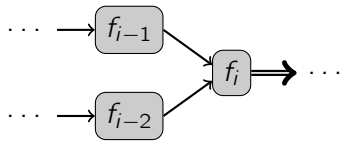


Figure 45: Unifying multiple streams into one

In `streams.io` package there is a `Join` processor that performs a union of the given input streams. This is realized by making this processor a sink node to which multiple processes can forward their data items. Though when executed on a distributed processing engine, this functionality can be simply mapped to the native `union` API.

5.7.5 ML Operations

Support for ML algorithms is one of the essential tasks discussed in this thesis. In section 3.3, several frameworks for distributed ML approaches have been introduces. Two concepts for designing both streaming and batch ML approaches through `streams` are illustrated here.

Streaming Consider the distributed streaming ML framework SAMOA. It utilizes a similar concept to build its job graph as Storm. SAMOA allows for wrapping customized processors next to its own distributed streaming ML algorithms. That is, the pipeline can contain several pre- or postprocessing steps and a learning algorithm. Using `streams` it is possible to define this job graph the same way as for Storm, Spark and Flink, but with the additional ML processors. In this case, the packages `streams-flink` and `streams-storm` are not required directly as SAMOA produces both Flink and Storm topologies. This can be accomplished like in the `streams-moa` package [54]. The code snippet below illustrates job definition where pre- and postprocessing steps are combined with a predictor, an evaluator and a learner:

```
<stream id="data" class="stream.io.CsvStream" url=""/>
<process input="data">
    <PreprocessingProcessor />
    <stream.learner.Prediction learner="learner" />
    <stream.learner.evaluation.PredictionError />
    <samoa.learners.classifiers.trees.VerticalHoeffdingTree id="learner" .../>
    <PostprocessingProcessor/>
</process>
```

Batch If the desired ML approach is not applicable in a streaming environment such as SVM, Random Forest or PCA, a batch processing approach is required. During this thesis the definition and the execution of a batch processing job was not realized. An interesting approach is presented in [58]. With an abstract concept of batch processing on top of `streams`, the definition of a pipeline like in Flink or Spark is simple. Considering a Flink pipeline in Figure 24, it can be designed with the `streams` framework as following:

```
<stream id="data" class="stream.io.CsvStream" url="..."/>
<batchProcess input="data">
    <PreprocessingProcessor />
    <flink.transformer.StandardScaler .../>
    <flink.transformer.PolynomialFeatures .../>
    <flink.predictor.MultipleLinearRegression .../>
    <stream.learner.TrainModel id="learner" .../>
    <stream.learner.Prediction learner="learner"/>
    <PostprocessingProcessor/>
</batchProcess>
```

To achieve this, the packages `streams-flink` and `streams-spark` can be extended to support batch processing. Afterwards, ML libraries such as FlinkML and Spark's MLlib can be utilized in a way presented above.

5.8 Summary

The abstract way of the job definition that is used in the `streams` environment matches the data flow graphs generated by the APIs of the distributed processing frameworks. In this section it was presented how the concepts of `streams` are mapped into distributed environments. This way all the implemented processors so far can be utilized to process Big Data on distributed clusters. Nevertheless not every feature and concept of the distributed processing frameworks can be used so far. Features like stateful processing, fault tolerance and windowing were introduced and only the ways of implementing them were discussed. Additionally, Spark and Flink support some aggregators like `sum`, `min` and `max` that can be simply utilized using XML configuration language of `streams` with the result inserted into `streams.Data` map.

6 Evaluation

Modern distributed processing frameworks claim to achieve higher performance throughput than their competitors. Various comparisons have measured the performance of distributed stream computing frameworks introduced in section 2.4. In most cases, the results differ probably depending on the structure and complexity of the data, different tuning settings and job's pipeline itself. In [59], the authors were able to show that Storm is still capable to compete with them and performs even better than Spark. Another comparison claims Flink's win in performance over Spark[43]. Further comparisons can be found proving better performance of each framework. In most cases simple textual data like tweets was analyzed (for example, word counting). Those comparisons lack in processing more complex data. Therefore, at first similar textual data analysis is used for the evaluation of the implemented software. Then performance differences are measured using data with more complex event items.

Preliminary Evaluation Apache Flink provides a Storm compatibility layer which allows for executing Storm's topologies on top of Flink. This way, for the testing purposes simple `streams` processes were executed on Storm and Flink before `streams-flink`'s development has been started. In both cases, `streams-storm` was utilized to construct the topology. The test process marks each incoming item with *A* or *B* alternatingly, adds some of them to a queue and writes down to a file. An excerpt from the process declaration looks as following:

```
<process input="data">
    <streams.util.AlternateAB/>
    <stream.Enqueue .../>
    <stream.io.JSONWriter .../>
</process>
```

The results are presented in Table 5. The throughput of data items clearly increased in Flink against Storm. Only in case of the processor `JSONWriter` Storm is much faster than its contender. In the end, the job deployed to Flink through the Storm compatibility layer reduced the runtime and increased data throughput by around 40 %.

Both topologies were executed locally on MacBook Pro Late 2012 with 2.5 GHz Intel Core i5 and 8 GB 1600 MHz DDR3 memory. All tests were set to use a single core with no further system tuning.

These first promising results motivated the further development and the more intense evaluation. In this thesis `streams` framework was presented as an abstraction layer to define data processing pipelines for different kind of engines. The new packages `streams-spark` and `streams-flink` were implemented to execute `streams` jobs on top of Spark and Flink

[43] http://www.kdnuggets.com/2015/11/fast-big-data-apache-flink-spark-streaming.html

	Throughput in items/sec	
	`storm`	`flink`
stream.util.AlternateAB	1 119 509.88	2 558 339.22
stream.demo.Enqueue	8359.96	13 248.86
stream.io.JSONWriter	31 596.79	8927.24
Total	5649.33	9105.25
(in seconds)	265.52	164.74

Table 5: Comparison of throughput and computation time of a Storm's topology executed directly in Storm and using a Storm compatibility layer in Flink.

distributed engines. Furthermore, the support for Storm within `streams-storm` was extended. With these tools it is possible to perform the same task on multiple platforms and compare the performance. Section 6.1 shows the configuration for the three clusters running natively Storm, Spark and Flink. Afterwards in section 6.2 various data and processing jobs are discussed together with the results of distributed execution.

6.1 Cluster Configuration

For the evaluation of the implemented features, the native clusters for Storm, Spark and Flink are used. This can be replaced with other more advanced solutions such as YARN or Docker. In this thesis, the jobs are executed on the native clusters to reduce the overhead and possible errors while preparing YARN or Docker cluster.

For comparable results four physical machines with the same hardware is used. As shown in Table 6, all the machines provide 6 cores and 32 GB of memory. The graphics card was not used for any of the tasks. This hardware configuration ensures comparable results of parallel job execution on all of the utilized machines. Additionally, using a shared folder for all the machines simplifies sharing the configuration files for each cluster.

In the following the basic configuration details are described for each of the utilized frameworks.

ls8ws010	129.217.30.171	Dell Precision Tower 7810 **CPU**: 6 Core Intel(R) Xeon(R) E5-2609 v3 @ 1.90 GHz **GPU**: Nvidia GK110GL Tesla K20c **RAM**: DDR4 32 768 MB **Kernel**: Linux 2.6 64
ls8ws009	129.217.30.199	
ls8ws008	129.217.30.198	
ls8ws007	129.217.30.197	

Table 6: Machines used for testing purposes of Storm, Flink and Spark Streaming

Storm Configuration Storm cluster requires three essential components: ZooKeeper, Nimbus and multiple Supervisors. All the configuration is contained in the `conf/storm.yaml` file and is used by Nimbus and Supervisors. Setting up the Storm cluster consists of the three steps that need to be performed in the following order:

1. **ZooKeeper** supports Storm's coordination between nimbus and supervisors inside the cluster. In most cases, one single ZooKeeper instance is sufficient as it is only used for coordination and not for message passing. For a higher level of fault tolerance one may used multiple ZooKeeper instances.

2. With a running ZooKeeper instance, the master node initializes the **Nimbus** daemon thread. The Nimbus is responsible for receiving new job requests and managing the currently running jobs. It can be started with the command `bin/storm nimbus` that is located in the downloadable Storm package. An optional, but essential part for tuning and debugging the cluster is the web dashboard. It can be initialized together with the Nimbus with the command `bin/storm ui`.

3. Each of the **Supervisors** can be registered at the running Nimbus instance with the command `bin/storm supervisor`.

For the above steps to work, some general options need to be set in the previously mentioned `conf/storm.yaml` file. In the following only five options are presented.

As seen above, for the first step ZooKeeper servers have to be named. In case of a rather small cluster which does not require high availability, only one ZooKeeper instance is instantiated:

```
storm.zookeeper.servers:
        - "ls8ws007.cs.uni-dortmund.de"
```

For the initialization of the Nimbus daemon and successful registration of the Supervisors, each component needs to know the master node. It is possible to initialize multiple Nimbus instances for high availability systems:

```
nimbus.seeds: ["ls8ws007.cs.uni-dortmund.de"]
```

The number of slots that defines the level of parallelization is adjusted by setting different ports in the configuration:

```
supervisor.slots.ports:
        - 6700
        - 6701
```

The above example will create two slots for that Supervisor daemon.
The memory usage is assigned depending on the following entry:

```
worker.childopts: "-Xmx1024m"
```

That is, the Supervisor will override the settings of the JVM and receive the defined amount of memory. The last important setting to be mentioned here is the path to a destination where each Storm component can store it settings, logs, jars and so on:

```
storm.local.dir: "storm-local"
```

Storm allows for more configuration which are not handled here.

Flink Configuration In contrast to Storm, Flink does not require a ZooKeeper instance except for a high availability mode. After configuring Flink by editing `conf/flink-conf.yaml` and adding all taskmanagers to be initialized to `conf/slaves`, Flink cluster can be started by running the following script:

```
bin/start-cluster.sh
```

The cluster is extendable with further jobmanagers and taskmanagers. These instances can be added or stopped with the following scripts:

```
bin/jobmanager.sh (start cluster) | stop | stop-all
bin/taskmanager.sh start | stop | stop-all
```

The configuration is similar to Storm. There are five important options to be considered for configuration:

```
jobmanager.rpc.address: "ls8ws007.cs.uni-dortmund.de"
parallelism.default: 1
taskmanager.numberOfTaskSlots: 1
taskmanager.heap.mb: 2048
taskmanager.tmp.dirs: /tmp
```

The most essential setting is the address of the master node as every component, `jobmanager` and `taskmanager`, needs it for initialization and registration. The other parameters have defaults values that need to be configured in order to achieve high performance. The level of parallelism set in the configuration is applied if the task does not specify its own value of parallelism. For a taskmanager it is important to set the number of task slots which corresponds to the CPUs used for parallelizing the operators or functions. Furthermore, the amount of memory for each taskmanager is split among all the slots. Hence, it is preferable to use as much memory as possible leaving enough memory for the operating system. As the last option listed above, the path to a directory is set where all the logs, jars etc. are saved for each taskmanager.

The other configuration file used for automatic cluster setup with the aforementioned script is `conf/slaves`. Its structure is very simple and only requires an IP address of each taskmanager per line:

```
129.217.30.171
```

```
129.217.30.199
129.217.30.198
```

Spark Configuration Spark's configuration is more similar to that of a Flink's cluster as no ZooKeeper's instance is required except for the high availability mode. All the configuration is contained in `conf/spark-env.sh` file. Using the following single call of a script is enough to start a standalone Spark cluster:

```
./sbin/start-all.sh
```

This script initializes the master node and additionally all the worker nodes which are listed in `conf/slaves`. The file containing the list of slave (worker) nodes only requires a single IP address of the machine per line:

```
129.217.30.171
129.217.30.199
129.217.30.198
```

Later on, the cluster can be extended with further worker nodes by using another script:

```
./sbin/start-slave.sh <master-spark-URL>
```

Although the slave can be configured using the passed attributes, all the option can also be set by editing `conf/spark-env.sh` file.

```
SPARK_MASTER_HOST=129.217.30.197
SPARK_WORKER_CORES=1
SPARK_WORKER_MEMORY=2g
```

The number of cores and memory for a worker is the upper limit of the available CPUs and memory for the executors that run on that worker node. Additional tuning of a wide variety of options such as number of executors with their CPU and memory requirements is done by editing the file `conf/spark-defaults.conf` or by adding command-line parameters while submitting the job to the cluster.

Summary

Configuring the standalone cluster in all the presented frameworks follows the same concept. Flink and Spark reduce the overhead by providing a script to start the whole cluster with one command-line call. At the same time, Storm requires manual start of each component. It is advisable to run every component under supervision. A supervisor framework monitors the services and among other tasks is able to restart failed service. Furthermore, Storm is the only framework that requires a running ZooKeeper server for the cluster. From the other point of view, Spark presents a huge number of options that can be set in two different files. While these options on the one hand mean more freedom of choice for the user, on the other

hand it becomes difficult to overview all of them. Additionally, Spark adds another option to the parallelism tuning next to the workers, the executors and their settings. This becomes another conflicting point which should be considered carefully.

Summarizing the considerations above, Flink seems to provide the most simple cluster configuration. Although Spark has scripts for cluster setup, the huge amount of options may overwhelm the user. Storm provides a rather similar number of options as Flink, but requires a more manual setup of the cluster.

6.2 Data Stream Processing Applications

Using the implemented packages the `streams` jobs can be executed on the distributed processing engines. Four steps are required for this:

1. create a new or use an existing `streams` project
2. add a dependency for `streams-{`*`flink`*`,`*`spark`*`,`*`storm`*`}` for on the desired engine
3. package the resources with the mainclass responsible for the construction of the DAG
4. deploy / run this task on top of the desired distributed framework

For the third step, the following main classes are available for the different execution engines (set it in the pom.xml directly or define property and pass it as an parameter while packaging):

- `flink.deploy_on_flink`
- `spark.deploy_on_spark`
- `storm.deploy_on_storm`

All of the frameworks either provide a script for job submission including the compiled jar or are able to run the job after the jar has been transfered to the cluster's master node.

Tuning Spark Streaming As Spark requires more tuning parameters, the XML format has been extended to accept the following parameters and attributes for tuning Spark's performance. A Spark cluster executes the job on a number of worker machines where each machine can host one or multiple executors. An executor uses a predefined number of cores to process partitions simultaniously. Therefore, an executor performs the tasks in parallel and is also responsible for the in-memory storage for RDDs. In section 2.4.2, the equation to calculate the best value for block interval that determines the number of partitions has been introduced. This requires the knowledge about the available resources. One fact is that each receiver uses one core. Hence, only $\#allAvailableSparkCores - \#numberReceivers$ cores are used for processing the data. For this the additional properties *numberSources*,

partitionFactor, *sparkCores* and *spark-batch-interval* define the number of stream sources, partition factor, all the available Spark cores and the batch interval respectively. These parameters are shown in Figure 46.

When submitting a new job to the cluster, it is important to tune the number of executor instances, the number of cores and memory inside of each executor. As the processing time correlates with the number of partitions, in case less partitions are created than available cores, not all the possible computing power is used. Furthermore, using a number of partitions not multiple of usable cores will lead to a situation where some of usable cores will wait for other processes to finish. In `streams-spark`, it is sufficient to configure the number of available cores, the number of sources and the partitioning factor to create the appropriate amount of partitions.

Furthermore, when using various physical machines, it is important to provide the system with the right setting for the locality of the data. That is, the default settings may force processing all the data items only on the local physical machine with the remaining cluster being unused. Thus, sharing data from a source onto a different computing node needs adjusting the settings for locality. This can be done by passing the desired values to a job-submission script as followed:

```
--conf "spark.locality.wait=3s"   --conf "spark.locality.wait.process=1s"
--conf "spark.locality.wait.node=1s"   --conf "spark.locality.wait.rack=0s"
```

This way one is able to define the settings when the data can change the locality from *process* to *node* or even *rack* level.

Furthermore, one core is used by the driver program. Now when submitting the job the executor settings are provided as following:

```
--num-executors 3 --executor-cores 6 --executor-memory 24G
```

For a cluster with 6 cores in each of the three available machines the above configuration does not work out as it requests 18 cores with one core reserved by a driver. This fact makes tuning a bit more difficult.[44] In order to exploit as much processing power of the cluster as possible, multiple executors are initialized with one core each:

```
--num-executors 17 --executor-cores 1 --executor-memory 4G
```

This way 17 cores of the cluster are utilized running in single JVMs with 4 GB of memory each. For the experiments in this thesis, it is not essential to bundle multiple cores to an executor whereas other applications with more intense inter-process communication (e.g. using broadcast variables) might face performance drops.[45]

In contrast to a streaming framework like Storm or Flink, as a micro-batch job Spark loses some time for the initialization and deserialization of the job itself. This time can be examined in the web dashboard and is rather small, but should not be fully ignored.

[44] http://etlcode.com/index.php/blog/info/Bigdata/Apache-Spark-tuning-spark-jobs-Optimal-setting-for-exe

[45] https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

```
<container spark-batch-interval="1000"
                    spark-master-node="spark://129.0.0.1:7077">
        <property name="numberSources" value="1" />
        <property name="partitionFactor" value="1" />
        <property name="sparkCores" value="5" />
        ...
```

Figure 46: Tuning parameters for Spark Streaming required for calculating the appropriate number of partitions.

6.2.1 Twitter Stream

To start off with a comparison of `streams`' runtime with the various frameworks a simple example like word-counting is constructed for a single machine execution. For this purpose the clusters are configured using the above mentioned settings, but only using one single machine. The material has been prepared within the ongoing research project VaVel (Variety, Veracity, Value).[46] This project is motivated by using urban data to improve situation aware transportation. To achieve this aim, multiple heterogeneous urban data streams should be combined and analyzed for better trip planning. The goal is to combine the desired components into a general purpose framework. Liebig et al. [60] introduces such a framework based on `streams`. As `streams` is decoupled from the execution engine and the amount of data matches the Big Data definition, it is possible to process the data from VaVel project using any framework for distributed computation. This preliminary comparison uses a collection of tweets, which are read from a file, and the number of tweets containing predefined tags is counted. It was performed to be presented at a VaVel meeting in Dublin where the various distributed processing frameworks and the pipeline definition using `streams` were discussed.

The XML configuration for this task is shown in the Figure 47. Inside the stream element, the *limit* attribute forces the source to stop after the predefined number of read tweets. Although the source can theoretically be parallelized, this functionality has not been implemented for the `stream.io.CsvStream` stream component and hence, the number of copies is set to one. The process node is intended to make use of the *copies* attribute in contrast to the stream source. Inside the process node *process-tweets*, the analysis of the twitter stream is wrapped in to a `streams.Performance` processor list, which enables logging the performance values of the distributed parallel application to a server.

Measurements The one-machine-clusters were given 28 GB of memory and all the 6 cores were initialized. The processing task has been executed using all the presented distributed processing engines and `streams-runtime` where the number of parallel instances

[46] `http://www.vavel-project.eu/`

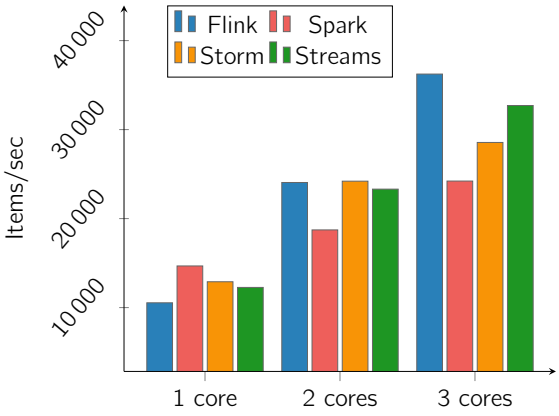
```
<container>
    <stream id="tweet-stream" class="stream.io.CsvStream"
                url="file:///home/share03/datensaetze/vavel/Twitter_IE/Twitter_IE.csv"
                limit="500000" copies="1"/>
    <process copies="1" input="tweet-stream" id="process-tweets">
        <streams.Performance host="129.217.30.198" port="6001" every="5000">
            <com.ibm.vavel.benchmarks.TwitterAnalysis textField="MESSAGETEXT"/>
        </streams.Performance>
    </process>
</container>
```

Figure 47: XML configuration for a data processing pipeline where tweets are read from a file and processed inside the process element. The performance is logged using the `streams.Performance` processor.

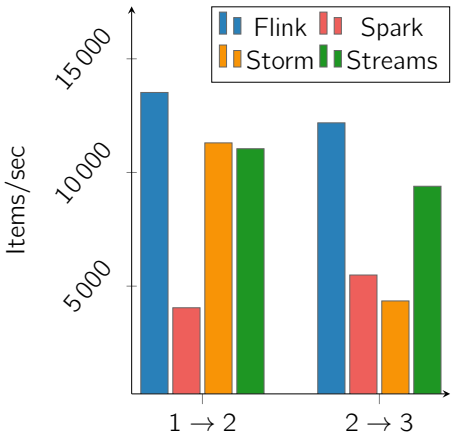
of the process node varied between one and three. Higher number of cores showed no further improvements and hence are not considered here. The reason is the source that is not parallelized and thus is limited by a maximum read speed of a single instance. The results can be viewed in Figure 48a. For this figure, the overall computing time of the application has been used to determine the throughput of items per second. Considering `streams-runtime`, the performance is increasing almost linearly. The same effect can be considered for Flink and Spark while Flink is increasing faster than Spark. Storm performs twice as good on the two cores, but does not hold the same improvement rate for three cores. These improvements can be viewed in Figure 48b.

Another interesting effect is pictured in Figure 48c where the average throughput of the processor is measured. `streams` outperforms every other framework followed closely by Flink. Even while Spark showed the best result in the one-core situation for the overall performance, the processor itself shows the worst result. Generally, the differences between the two figures might be explained with the *pull* concept of `streams`. That is, the process retrieves the next data item from the source after it has finished the computation. In contrast to this concept, in the other frameworks the source is continuously producing data items and *pushing* them into the system. This way, the delay between the end of processing the data item and the arrival of the next data item is reduced. Therefore, Flink is able to process the same number of data items faster than `streams` while its average processor performance is even slightly worse. The reduced average performance may be ignored at the price of the simplified job definition and the distributed execution which can not be done through `streams` itself.

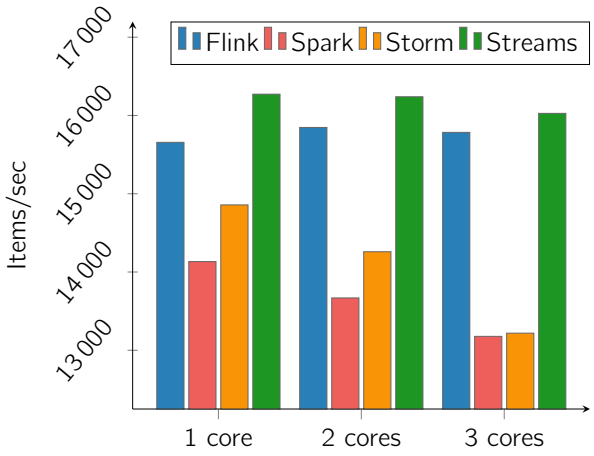
Tuning the Spark job turned out more difficult than it was assumed. In order to investigate this effect, the web dashboard provides helpful information. During this experiments the load balancing problems were observed as shown in Figure 48d. While for the single-core execution Spark simply uses one partition, in the multi-core execution one needs to tune the right amount of partitions to be created. Although the data should be split in 3 partitions



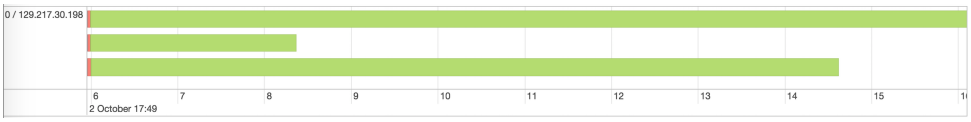
(a) Overall throughput performance of the different execution engines. The values are computed based on the application's computing time divided by the number of processed items.



(b) The improvements of the throughput performance while increasing the level of parallelism from 1 to 2 and from 2 to 3 cores.



(c) Average throughput performance of the single processor unit (word matching function).



(d) Detailed view on a single batch inside of Spark Streaming where the load is not balanced among the 3 available computing slots (cores).

Figure 48: Performance results of various distributed processing frameworks compared to `streams-runtime`.

over 3 cores, the computing time on each core differs a lot. That is, the data items are not distributed equally.

6.2.2 FACT Telescope Image Data

The observation of various galactic and extra-galactic sources in astronomy relies on the measured energy emitted by their origins. While there exist different techniques to measure that energy, one of it measures the Cherenkov light emitted by the air showers which is generated by the particles entering the atmosphere and interacting with its elements. FACT (First G-APD Cherenkov Telescope) produces images of the Cherenkov light captured by the telescope camera. Such an image contains 1440 pixels and is sampled at a very high rate (around 2000 MHz) [61]. In case the hardware triggers a possible shower, a series of images is written to disk for the analysis. This series is called *region of interest* (ROI). For the analysis ROI is considered as an event.

Not every particle can be used to detect its origin as hadronic particles may have interfered with electromagnetic fields. In contrast to hadrons, gamma particles are uncharged and hence do not change their direction. The aim of the research is hence to distinguish gammas from hadrons. To achieve this, the analysis pipeline consists out of the following general steps:

1. calibration, cleaning
2. feature extraction
3. signal separation
4. energy estimation

Based on the `streams` framework, `fact-tools`[47] was developed to simplify the analysis pipeline definition for researchers without specifying the underlying execution engine. This package contains domain specific functions implemented directly for the FACT data stream analysis. Figure 49 presents some of the functions applied in the analysis pipeline using `fact-tools`. Calibration and cleaning takes the most time as during these phases pixels of each single image have to be considered and investigated for possible events. While the most research is done in feature extraction and energy estimation, the first step consumes the most computing resources. Therefore, this analysis chain can be computed on a distributed framework to be able to match the data rate of the telescope in best case.

Process configuration and details For this experiment, the FACT pre-processing pipeline with the most time consuming steps from the data calibration and cleaning was used. A part of the XML configuration that was used is presented in Figure 50. At the beginning some

[47] `https://sfb876.de/fact-tools/`

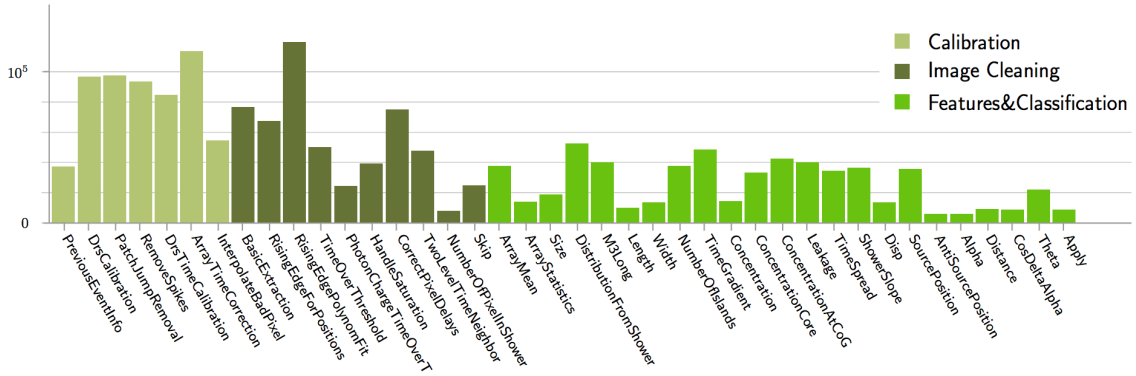


Figure 49: Processing time of some processors from `fact-tools` (log scale) [61]

properties and services are initialized. Then the `stream.IterateFilesStream` source is utilized which was implemented as a parallel stream source. That is, the stream component is initialized separately for each instance and retrieves the data items from different files. The data is streamed from the Fits files located in a HDFS cluster using the subcomponent of the stream element `fact.io.FitsStream`. For a real boost in processing files from the HDFS storage, it is advisable to run the application directly using YARN. The limit for the stream is set to 20 000 data items with around 17.2 GB size.[48] After that, the process node *fact-process* contains the processing steps and is parallelized using the *copies* attribute. The particular list of processors is not shown here due to its length. Finally, the amount of data to be processed was set to 150 000 such that the applications needs more time to execute the job. For this size of the transmitted data is around 129 GB.

Cluster setup For the following experiments three cluster configurations are used:

- 1 machine with 25 GB of RAM and 6 cores
- 2 machines with 50 GB of RAM and 12 cores
- 3 machines with 75 GB of RAM and 18 cores

Storm

Table 7 presents the results from processing FACT data using `streams` on top of Storm. The first setting uses one machine, one source and 6 slots for processing the stream. One slot is reserved almost the whole time for the spout that retrieves the data from HDFS cluster. The latency raised up to 30 s. Some of the data items were marked as failed as the default

[48] Limited by current implementation of the distributed parallel source. In case of three sources, a limit of 19998 items is set to be able to split 20 000 items onto three sources.

```
<container id="performance-test" copies="1">
    <property name="infile" value="hdfs://ls8cb01.cs.uni-dortmund.de:9000/FACT/simulation/" />
    <property name="auxFolder" value="file:/home/egorov/aux/" />
    ...
    <service id="auxService" class="fact.auxservice.AuxFileService" auxFolder="${auxFolder}" />
    <service id="calibService" class="fact.calibrationservice.ConstantCalibService" />

    <stream id="fact" class="stream.IterateFilesStream" url="${infile}" ending=".fits.gz"
                copies="1" limit="20000">
        <stream id="_" class="fact.io.FitsStream"/>
    </stream>

    <process id="fact-process" input="fact" copies="6">
    ...
    </process
</container>
```

Figure 50: An excerpt of the FACT processing job defining the services and a parallel stream source. The processing steps are not included due to the length of the list.

topology message timeout after which the data item is considered failed is also 30 s. Every failed message was transmitted to the topology again.

Adding a second machine with another 6 free slots gained almost twice the performance as originally. With 12 processing slots Storm reduced the latency down to 190 ms. This time no message failed. Running the same configuration with 2 parallel spout did not doubled the performance. A small improvement of around 5 % was measured.

As awaited, adding another same configured machine to the setup improves the throughput of the system. Utilizing two sources on three machines is still slower than three sources. As presented in the overview table, three machines with three spouts gain 65.6 % of performance improvements and reduce the computation time radically. The measured topology latency was around 3 s.

Finally, the amount of data to be processed was set to 150 000 such that the applications needs more time to execute the job. Although the processing speed slightly increased to 163.18 items/s, the latency of the topology was continuously growing and seemed to stay constant at 14 s. Processing this amount of data consumed 919.24 s. Some of the messages failed, but still all 150 000 data items were processed as presented in Figure 51.

Spark Streaming

The size and complexity of the FACT data compared to the short textual data in a Twitter stream is rather big. It becomes a matter of smart tuning the whole system to prevent the growing delays which lead the system to a possible crash as the unprocessed data stays in memory (or is written down to a storage). Backpressure worked fine with the simple text messages from the previous section, but did not seem to change something when scheduling

Setup	Sources	Time	$\frac{Items}{sec}$	Improvement	Latency
1 machine, 6 cores, 20k	1	362.87	55	–	
2 machines, 12 cores, 20k	1	193.58	103	46.6 %	200 ms
2 machines, 12 cores, 20k	2	185.79	108	48.8 %	8100 ms
3 machines, 18 cores, 20k	2	153.32	130.45	57.8 %	1100 ms
3 machines, 18 cores, 20k	3	125.20	159.73	65.6 %	3100 ms
3 machines, 18 cores, 150k	3	919.24	163.18		11 000 ms

Table 7: Comparison of different setups for Storm running FACT processing pipeline to analyze telescope events. The number of distributed parallel sources is varying next to different setups. The improvement is computed compared to 1-machine setup.

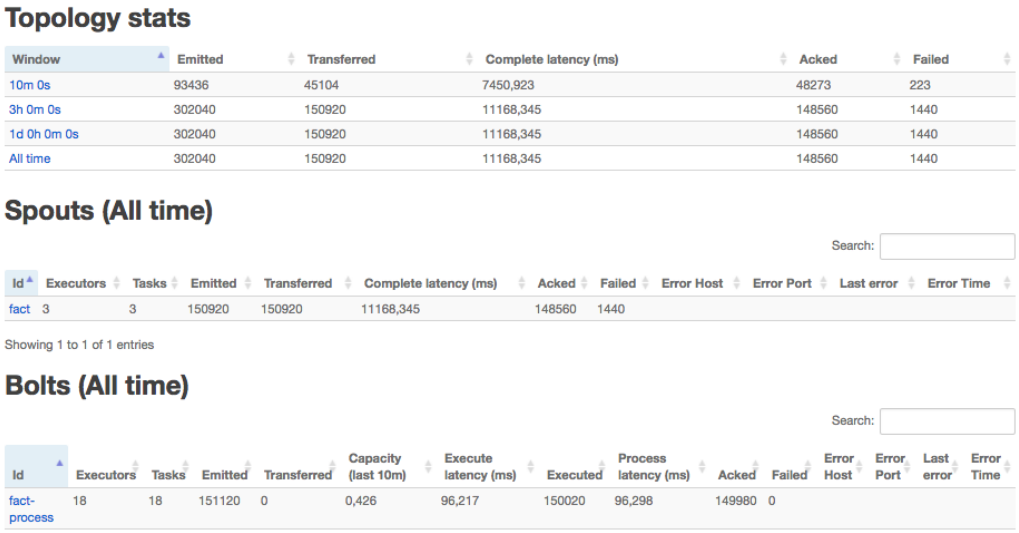
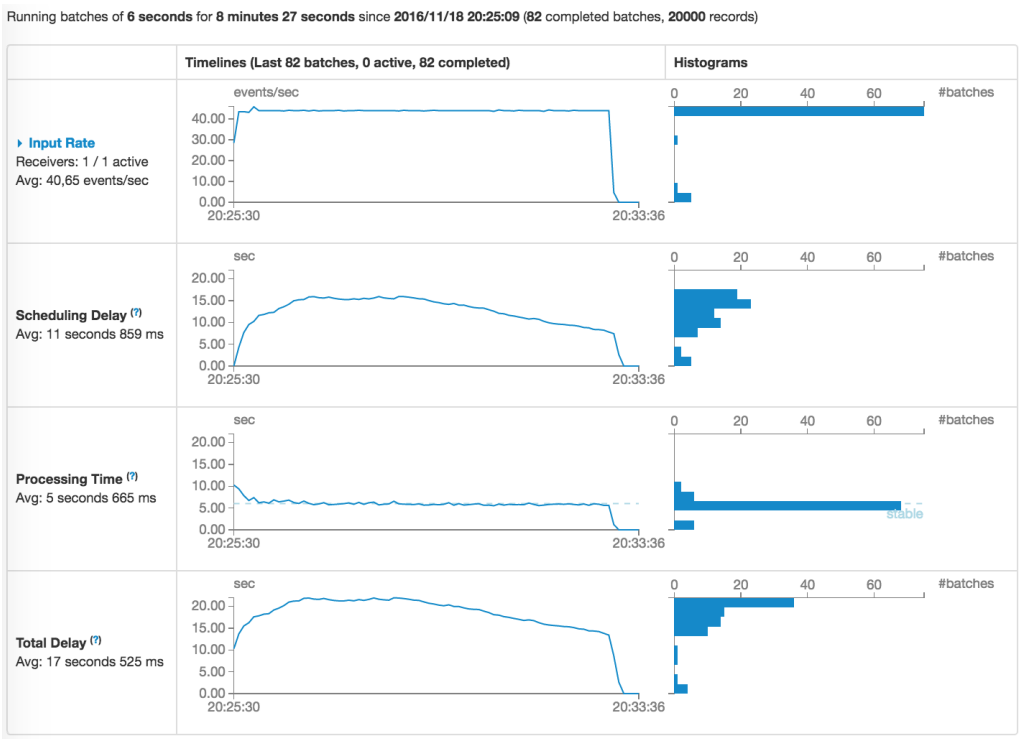


Figure 51: Storm processing 150 000 FACT data items

and delay time were growing rapidly using FACT data. The maximum receiver rate was limited to prevent the overload of the system.

Table 8 presents the results over various cluster setups. During all the runs only those results were recorded where the delay time of the system recovered after some time. The result of a single machine setup of 498 s was achieved using the following settings: `maxrate` of 47 items/s for each receiver, the batch interval of 6 s with 4 partitions per core. The delays and throughput of the system can be viewed in Figure 52a. The processing time for each batch is almost optimal and the scheduling delay recover from the early increase. Considering a single batch job in Figure 52b, the above configuration achieves fair partition processing among the four available cores. Though for every partition deserialization of the task (marked in red) consumes some time. This has an impact on the overall performance.



(a) Processing FACT data with Spark Streaming on a single machine (batch interval 6 s, receivers' maxrate 47 items/s).

(b) Processed partitions on a single machine. From initially 6 cores one is used by the driver program and another one by the receiver.

Figure 52: Spark Streaming dashboard showing statistics of processing FACT data.

Setup	Sources	Time	$\frac{Items}{sec}$	Improvement
1 machine, 6 cores, 20k	1	498	40.16	–
2 machines, 12 cores, 20k	1	312	64.10	37.35 %
2 machines, 12 cores, 20k	2	220	91	55.82 %
3 machines, 18 cores, 20k	3	180	111.11	63.86 %
3 machines, 18 cores, 150k	3	1320	113.63	

Table 8: Comparison of different setups for Spark Streaming running FACT processing pipeline to analyze telescope events. The number of distributed parallel sources is varying next to different setups. The improvement is computed compared to 1-machine setup.

The extension of the setup with a second machine and using two parallel sources doubles the throughput of the system and halves the duration of the application. This is achieved by keeping *maxRate* at the same level and adjusting the locality values such that the data from the receiver on the machine A is not transmitted to the machine B. In contrast to this result, the same application using only one receiver performs worse due to the transmission costs and the limited receiver rate. Finally, adding the third machine to the setup pushes the performance further reducing the duration up to 180 s.

In contrast to Storm, during the experiments it was not possible to tune the system such that it performs better on three machines with two sources than on two machines with two sources. Furthermore, as Spark reserves one core for the driver, only 17 cores were available for processing at most. Each receiver (source) utilizes a core and hence in the three-machines setup only 14 cores are used for processing. Comparing the one-machine and three-machines setups the throughput of the system is improved by a factor of 2.7. The improvement could have been higher if Spark Streaming would distribute the parallel sources on the different machines. Unfortunately, three sources were spread over two machines and there is no direct option to force it.

Flink

Configuring the parallelism of a Flink job turned out simpler than in Spark. After setting up the cluster, using the *copies* attribute is sufficient to improve the system performance. The first setup utilizes the resources of a single machine. Processing all the 20 000 data items took 289.65 s (average: 69.05 items/s). The addition of a second machine reduced the time needed to process 20 000 data items down to 216.67 (average: 92.31 items/s). This is an increase in the items processed per second and logically a descrease in the duration of the application. Although the resources were doubled, the observed performance improvement in the application's duration time was around 24 %. In this setup the data was distributed from one machine receiving the data items across the two worker machines. Receiving and

transmitting of a data item uses the same data connection and hence forwarding a data item from a receiver machine A to the second machine B reduces the bandwidth. Hence, the second source was added to retrieve the data items in parallel. In this new setup Flink automatically executes parallel sources on different machines. The processing time clearly goes down to 165.64 s (average: 120.75 items/s). By adding another machine with another parallel source to this setup improves the processing time up to 118.92 s as can be seen.

Setup	Sources	Time	$\frac{Items}{sec}$	Improvement
1 machine, 6 cores, 20k	1	289.65	69.05	—
2 machines, 12 cores, 20k	1	216.67	92.31	25.20 %
2 machines, 12 cores, 20k	2	165.64	120.75	42.81 %
2 machines, 12 cores, 20k, rescaled	2	153.53	130.27	46.99 %
3 machines, 18 cores, 20k	2	128.32	155.86	55.70 %
3 machines, 18 cores, 20k, rescaled	2	120.41	166.10	58.43 %
3 machines, 18 cores, 20k	3	118.92	168.16	58.94 %
3 machines, 18 cores, 20k, rescaled	3	113.69	175.91	60.75 %
3 machines, 18 cores, 150k, rescaled	3	884.59	169.57	

Table 9: Comparison of different setups for Flink running FACT processing pipeline to analyze telescope events. The number of available cores and distributed parallel sources is varying. The improvement is computed compared to 1-machine setup.

Using the `rescale()` API, the performance of the system is up to 2.5 times higher which is summarized in Table 9. In the setup with two or more parallel sources on the multiple computing nodes, `rescale()` forces local round-robin partitioning and reduces the transmission to other physical machines. In case of three computing nodes, the resulting throughput raised up to 175.91 items/s with the reduced computing time of 113.69 s. Using this most powerful Flink setup, processing 150 000 data items consumed 884.59 s. Figure 53 presents Flink's dashboard with the results of the last experiment where stream sources are spread over three different machines.

When adding the third source to the setup, almost no performance increase can be measured. This effect can have various reasons. On the one hand, the backpressure in Flink can be protecting the system well from too high bursts of data. Otherwise, it is possible that using three parallel sources to transmit the data over the network reaches its limits.

6.3 Summary

Distributing the processing of the Twitter stream was rather simple due to the structure and size of the data items themselves (light-weight text messages). The image data produced by

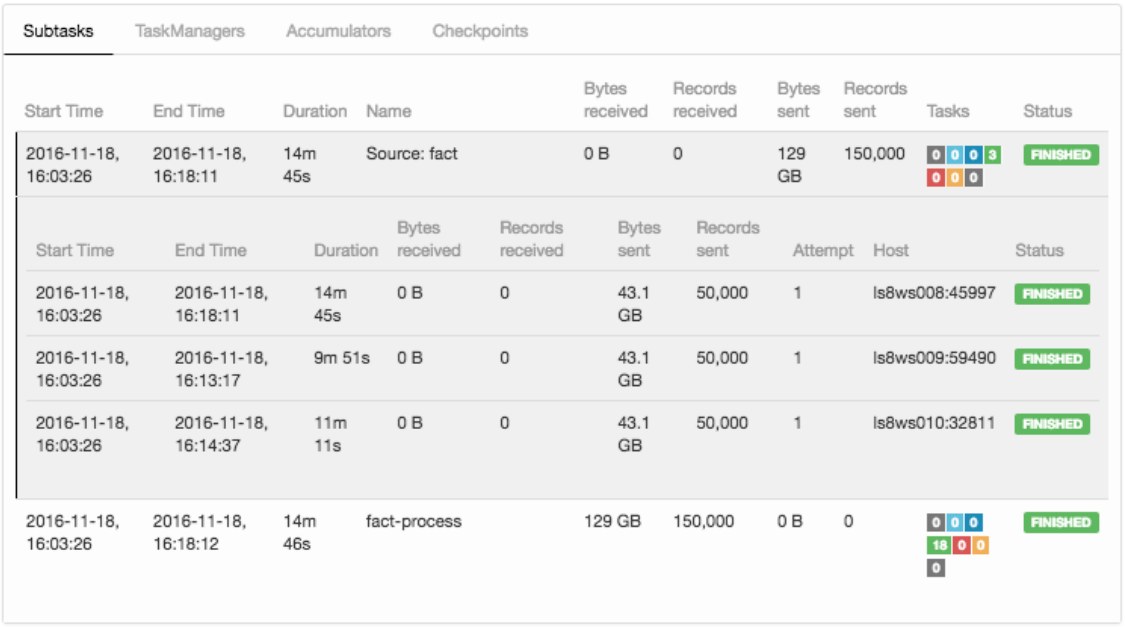
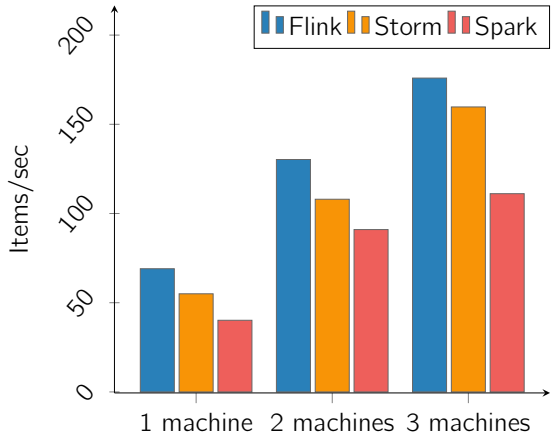


Figure 53: Running through FACT pre-processing pipeline (data calibration and cleaning) inside of Flink. The stream was limited to 150 000 items.

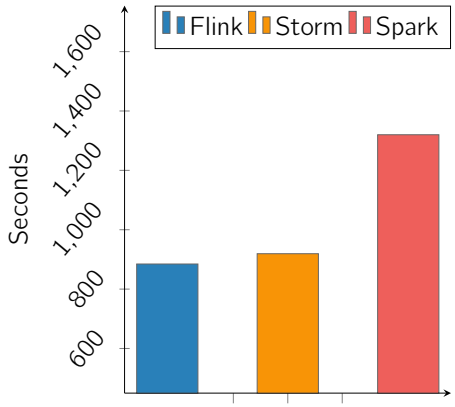
FACT raises the computation time for the processing. Additionally the costs of transmitting the data to another physical machine need to be considered which induce strong conditions for data locality. This is achieved by holding the stream source and the processed data on the same worker node.

The throughput of the system was increased by the factors of 2.8, 2.7 and 2.5 with Storm, Spark and Flink respectively. As seen in Figure 54b, Flink was always able to perform preprocessing of the same amount of FACT data faster than Spark and Storm. While executing the job Spark reserves one core for the driver program. Although this is ignorable in a case of cluster with many machines with hundreds of CPU cores, for the experiments in this thesis the difference to other frameworks was measurable and makes it difficult to compare the results. Nevertheless, Spark performed on a three-machine setup as good as Storm using two machines only.

Due to a wide range of tuning parameters in Spark, it is difficult to state whether its performance can be increased further. Hence, when it comes to tuning the system for best performance, Spark can not hold against Flink and Storm. While a Flink job performed the results reported above withing the first run, Spark required multiple runs to distinguish the right values for *maxRate*, *batch interval*, data locality and number of partitions. During the experiments with the textual data using one partition per core in Spark Streaming showed bad load balancing as presented in Figure 48d. With this knowledge, for the experiments on FACT data a higher value of partitions was chosen and improved Spark performance radically.



(a) Comparison of Flink, Storm and Spark processing 17 GB of FACT data. Higher values are better.



(b) Comparison of Flink, Storm and Spark processing 129 GB of FACT data. Lower values are better.

Figure 54: Evaluation of Flink, Storm and Spark on the image data from FACT telescope

One approach was originally to split the FACT pipeline into several steps (*processes*) with different level of parallelism. Unfortunately, this approach was withdrawed as during the data correction and feature extraction process a lot of data is produced and added to the passed data item object. In case of the two following steps being executed on the different machines, this means transmitting original and newly calculated data between the nodes. A high decrease in the throughput performance was detected during several experiments. This should be investigated deeply in the future work.

Following the above considerations, Flink is able to provide the best performance values in both experiments almost no tuning overhead. Storm delivers results close to those of Flink and also requires no advanced adjustments of the system. Spark as a batch framework primarily needs heavy tuning of various parameters to achieve a comparable performance.

7 Summary and Conclusion

With the steadily increasing volume, complexity and velocity of the data the definition of Big Data is also evolving continuously. For the Internet of Things millions of devices and sensors provide streaming data to be processed and analyzed in real-time. As this task becomes difficult in terms of resourse constrained environments, various frameworks for distributed processing were introduced during the last decade for improving the applications parallelism. While the existing frameworks continue improving its performance, new contenders arise. Each of them provide different API for building distributed tasks.

The aim of this thesis was to simplify the job definition for different distributed processing engines using an abstract layer on top of those frameworks. Therefore, at first in section 2 the distributed systems and CAP theorem were discussed before an overview of three distributed stream processing frameworks was given. As analyzing the data stream is one of the most challenging tasks, section 3 presented a minor survey of available distributed machine learning frameworks.

The `streams` framework introduced in section 4 provides an abstract job definition layer that is decoupled from the execution engine. This concept was used to develop new packages `streams-spark` and `streams-flink` as presented in section 5. They are both based on the concepts of the existing `streams-storm` package that was updated and extended during this thesis. These extensions allow to transform `streams` jobs into native Spark, Flink and Storm distributed tasks. Finally, in section 6 the implemented software was evaluated using two different data sets.

Conclusion During this thesis it was of great interest to figure out which framework is capable to provide high performance while being able to analyze the data stream. The features of the discussed distributed processing frameworks are almost identical and hence no framework stands out with a very unique functional property. Though, there is a big difference when considering the libraries built for each of the engines. In this thesis, particularly ML libraries were inspected. Spark definitely has a much longer list of supported ML algorithms for the distributed environment.Nevertheless, Flink and Storm can be combinated with SAMOA for streaming ML approaches. On the other side, the results of a streaming data processing pipeline indicated Flink as a clear winner in terms of performance. Storm delivered values close to those of Flink. Tuning the performance of Spark Streaming turned out as a difficult task that need to be performed for every new distributed setting and data set to be processed. Flink and Storm required almost no tuning except the basic cluster configuration. Therefore, it is possible that Spark Streaming is capable of achieving higher performance values with more time invested into tuning the system.

In the end, a framework has to be chosen driven by the application task. Spark provides more ML capabilities while Flink and Storm process data streams much faster. Using `streams`

and the software developed during this thesis switching between the different frameworks was simplified. Hence, a possible combination is using Spark with MLlib for training a model and Flink or Storm for real-time requests using `streams` as an abstract job definition layer.

Future Work The core part of this thesis was to ensure mapping basic `streams` components onto various distributed processing frameworks. This way, almost every `streams` job can be executed on Storm, Flink and Spark Streaming. In section 5.7, several not implemented features such as stateful processing, windowing and ML capabilities were discussed. Additionally, the way to add their support was presented. While streaming ML is a good starting point, the next step is to introduce the definition of batch processing jobs through streams for Spark and Flink. As the concept of representing a job using DAG is the same for batch and streaming, this is an achievable goal.

Furthermore, a more intense usage of the native API of the distributed frameworks might help improve the performance. Spark and Flink both support functions such as `filter`, `reduce`, `union` etc. To achieve this, the current scheme of mapping `process` to a native function has to be changed to wrapping each single `processor`. Although it is possible to group processors within a `<process ...>` element, customizing each processor's parallelism allows more finegrained tuning. This topic should be investigated deeply in the future.

References

[1] C. Bockermann and H. Blom, "The streams framework," TU Dortmund University, Tech. Rep. 5, 12 2012.

[2] L. Mearian, "World's data will grow by 50x in next decade, idc study predicts," Jun. 2011. [Online]. Available: http://www.computerworld.com/article/2509588/data-center/world-s-data-will-grow-by-50x-in-next-decade--idc-study-predicts.html

[3] IDC, "The digital universe of opportunities: rich data and the increasing value of the internet of things," Apr. 2014. [Online]. Available: http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

[4] H. Weinzierl, "New digital universe study reveals big data gap: Less than 1% of world's data is analyzed; less than 20% percent is protected," Dec. 2012. [Online]. Available: http://www.emc.com/about/news/press/2012/20121211-01.htm

[5] M. Kanellos, "Amount of data created annually to reach 180 zettabytes in 2025," Mar. 2016. [Online]. Available: https://whatsthebigdata.com/2016/03/07/amount-of-data-created-annually-to-reach-180-zettabytes-in-2025/

[6] D. Tam, "Facebook processes more than 500 tb of data daily," Aug. 2012. [Online]. Available: http://www.cnet.com/news/facebook-processes-more-than-500-tb-of-data-daily/

[7] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, "A survey of open source tools for machine learning with big data in the hadoop ecosystem," *Journal of Big Data*, vol. 2, no. 1, pp. 1–36, 2015.

[8] C. Wu, R. Buyya, and K. Ramamohanarao, "Big data analytics = machine learning + cloud computing," *CoRR*, vol. abs/1601.03115, 2016.

[9] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, *Large-Scale Parallel Collaborative Filtering for the Netflix Prize*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–348.

[10] G. Le Lann, "Distributed systems-towards a formal approach." in *IFIP Congress*, vol. 7. Toronto, 1977, pp. 155–160.

[11] J. Dean and S. Ghemawat, "Map-reduce: Simplified data processing on large clusters," *D OSDI IEEE*, vol. 51, 2004.

[12] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[13] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, 2000.

[14] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[15] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[16] J. Kreps, "Questioning the lambda architecture," https://www.oreilly.com/ideas/questioning-the-lambda-architecture, 07 2014.

[17] Spotify, "How spotify scales apache storm," https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/, 01 2015. [Online]. Available: https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/

[18] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[19] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.

[20] C. C. Aggarwal, *Data streams: models and algorithms*. Springer Science & Business Media, 2007, vol. 31.

[21] N. Rivetti, Y. Busnel, and L. Querzoni, "Load-aware shedding in stream processing systems," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '16. New York, NY, USA: ACM, 2016, pp. 61–68.

[22] "The evolution of storm at yahoo and apache," http://yahoohadoop.tumblr.com/post/98751512631/the-evolution-of-storm-at-yahoo-and-apache, 09 2014. [Online]. Available: http://yahoohadoop.tumblr.com/post/98751512631/the-evolution-of-storm-at-yahoo-and-apache

[23] "History of apache storm and lessons learned," http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html, 10 2014. [Online]. Available: http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[25] Reynold Xin, "Apache spark officially sets a new record in large-scale sorting," https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html, nov 2014.

[26] R. Xin, P. Deyhim, G. A., X. Meng, and M. Zaharia, "Graysort on apache spark by databricks," http://sortbenchmark.org/ ApacheSpark2014.pdf, 2014.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 15–28.

[28] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*, 2012.

[29] G. Maas, "Tuning spark streaming for throughput," Dec. 2014. [Online]. Available: http://www.slideserve.co.uk/download/documents/tuning-spark-streaming-for-throughput-virdata

[30] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[31] Z. Prekopcsák, G. Makrai, T. Henk, and C. Gaspar-Papanek, "Radoop: Analyzing big data with rapidminer and hadoop," in *Proceedings of the 2nd RapidMiner community meeting and conference (RCOMM 2011)*. Citeseer, 2011, pp. 865–874.

[32] A. K. Koliopoulos, P. Yiapanis, F. Tekiner, G. Nenadic, and J. Keane, "A parallel distributed weka framework for big data mining using spark," in *2015 IEEE International Congress on Big Data*, June 2015, pp. 9–16.

[33] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*. Springer, 2000, pp. 1–15.

[34] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[35] A. Bifet, "Mining big data in real time," *Informatica*, vol. 37, no. 1, 2013.

[36] J. Read, A. Bifet, G. Holmes, and B. Pfahringer, "Scalable and efficient multi-label classification for evolving data streams," *Machine Learning*, vol. 88, no. 1-2, pp. 243–272, 2012.

[37] E. Almeida, C. Ferreira, and J. Gama, "Adaptive model rules from data streams," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2013, pp. 480–492.

[38] J. Duarte and J. Gama, "Ensembles of adaptive model rules from high-speed data streams." *BigMine*, vol. 14, pp. 198–213, 2014.

[39] A. Bifet, G. Holmes, B. Pfahringer, J. Read, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "Moa: a real-time analytics open source framework," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 617–620.

[40] M. Stolpe, "The internet of things: Opportunities and challenges for distributed data analysis," *ACM SIGKDD Explorations Newsletter*, vol. 18, no. 1, pp. 15–34, 2016.

[41] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, p. 281, 2007.

[42] H. Neeb and C. Kurrus, "Distributed k-nearest neighbors," 2016.

[43] G. Forman and B. Zhang, "Distributed data clustering can be efficient and exact," *ACM SIGKDD explorations newsletter*, vol. 2, no. 2, pp. 34–38, 2000.

[44] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler *et al.*, "Api design for machine learning software: experiences from the scikit-learn project," *arXiv preprint arXiv:1309.0238*, 2013.

[45] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "Mlbase: A distributed machine-learning system." in *CIDR*, vol. 1, 2013, pp. 2–1.

[46] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 34, pp. 1–7, 2016.

[47] G. De Francisci Morales and A. Bifet, "Samoa: Scalable advanced massive online analysis," *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 149–153, 2015.

[48] A. Bifet, S. Maniu, J. Qian, G. Tian, C. He, and W. Fan, "Streamdm: Advanced data mining in spark streaming," in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, Nov 2015, pp. 1608–1611.

[49] C. E. Seminario and D. C. Wilson, "Case study evaluation of mahout as a recommender platform," in *6th ACM conference on recommender engines (RecSys 2012)*, 2012, pp. 45–50.

[50] G. Kejela, R. M. Esteves, and C. Rong, "Predictive analytics of sensor data using distributed machine learning techniques," in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*. IEEE, 2014, pp. 626–631.

[51] KDNuggets, "R leads rapidminer, python catches up, big data tools grow, spark ignites," http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-spark.html, may 2015.

[52] David Whiting, "Data pipelines with apache crunch and java 8," https://developers.soundcloud.com/blog/data-pipelines-apache-crunch-java-8.html, jun 2016.

[53] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1235–1246.

[54] S. Rötner, "Behandlung von concept drift in zyklischen prozessen," Master's thesis, TU Dortmund, 2014.

[55] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "Moa: Massive online analysis," *Journal of Machine Learning Research*, vol. 11, no. May, pp. 1601–1604, 2010.

[56] N. Wulf, "Speicherung und analyse von bigdata am beispiel der daten des fact-teleskops," Master's thesis, TU Dortmund, 2013.

[57] Arun Mahadevan, "Windowing and state checkpointing in apache storm," https://community.hortonworks.com/articles/14171/windowing-and-state-checkpointing-in-apache-storm.html, feb 2016.

[58] M. Asmi, A. Bainczyk, M. Bunse, D. Gaidel, M. May, C. Pfeiffer, A. Schieweck, L. Schönberger, K. Stelzner, D. Sturm, C. Wiethoff, and L. Xu, "Pg594 - big data," TU Dortmund, Tech. Rep. 5, 10 2016, supervised by Katharina Morik, Christian Bockermann and Hendrik Blom.

[59] Yahoo, "Benchmark streaming computation engines at yahoo," https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, 12 2015. [Online]. Available: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[60] T. Liebig, N. Piatkowski, C. Bockermann, and K. Morik, "Dynamic route planning with real-time traffic predictions," *Information Systems*, pp. –, 2016.

[61] C. Bockermann, K. Brügge, J. Buss, A. Egorov, K. Morik, W. Rhode, and T. Ruhe, "Online analysis of high-volume data streams in astroparticle physics," in *Proceedings of the European Conference on Machine Learning (ECML), Industrial Track*. Springer Berlin Heidelberg, 2015.