An evaluation and performance comparison of different approaches for data stream processing

Charalampos Georgiadis
Abstract

An evaluation and performance comparison of different approaches for data stream processing

Charalampos Georgiadis

In recent years the demand of faster data processing and real-time analysis and reporting has grown substantially. Social networks, internet of things, trading are among others, use cases where data stream processing has a vital importance. This has led to the emergence of several distributed computing frameworks that can be successfully exploited for data stream processing purposes. This project aims to examine a number of them, their architecture and key features. First, all the open source frameworks were found and studied. Based on the approach they follow, two of them were selected to be further analyzed and presented in detail. In the final part of the project a telemetry data monitoring application was applied using both frameworks on a computing cluster. The aim of that experiment was to illustrate how those two different approaches would perform in terms of exploiting the cluster’s resources as the scale out.

Handledare: Konstantinos Vandikas
Ämnesgranskare: Salman Toor
Examinator: Mats Daniels
IT 16078
Tryckt av: Reprocentralen ITC
# Contents

1 Introduction .................................................................................................................. 7  
  1.1 Motivation and goals ............................................................................................... 7  
  1.2 Related work .......................................................................................................... 8  
  1.3 Report structure ..................................................................................................... 9  

2 State of the art solutions and frameworks ................................................................... 11  
  2.1 Background ........................................................................................................... 11  
  2.2 Criteria and background for the selection ................................................................ 13  
    2.2.1 Open Source frameworks ................................................................................. 13  
    2.2.2 Proprietary frameworks ..................................................................................... 14  
  2.3 Summary ................................................................................................................ 14  
  2.4 Frameworks under consideration ........................................................................... 17  
    2.4.1 Apache Flink ..................................................................................................... 17  
    2.4.2 Apache Samza ................................................................................................. 18  
    2.4.3 Apache Spark .................................................................................................. 19  
    2.4.4 Apache Storm ................................................................................................ 21  
    2.4.5 Esper ................................................................................................................. 23  
  2.5 Selection of major candidates under evaluation ....................................................... 23  

3 Apache Flink ................................................................................................................ 25  
  3.1 General architecture and process model ................................................................... 25  
  3.2 Jobs and scheduling ............................................................................................... 27  
    3.2.1 Job manager data structures ............................................................................ 27  
    3.2.2 Scheduling ....................................................................................................... 27  
  3.3 Iterative processing ............................................................................................... 28  
    3.3.1 Iterate operator ............................................................................................... 29  
    3.3.2 Delta Iterate operator ..................................................................................... 30  
  3.4 Memory management ............................................................................................ 31  
    3.4.1 Managed memory ......................................................................................... 32  
    3.4.2 Garbage collector ........................................................................................... 32  
  3.5 Optimization ......................................................................................................... 32  

4 Apache Spark ............................................................................................................. 33  
  4.1 General architecture and process model ................................................................ 33
List of Tables

Table 2.1: State-of-the-art frameworks overview ........................................... 16
Table 5.1: Network measurements from master node to worker nodes .......... 41
List of Figures

Figure 2.1: Word count example with MapReduce ........................................... 12
Figure 2.2: Flink’s workflow ................................................................. 18
Figure 2.3: Samza’s workflow ............................................................... 20
Figure 2.4: Spark’s workflow ................................................................. 21
Figure 2.5: Storm’s workflow ................................................................. 22
Figure 3.1: Different actors and their interaction in Flink ....................... 26
Figure 3.2: Flink’s software components stack ....................................... 27
Figure 3.3: Flink’s JobManager conversion from JobGraph to ExecutionGraph ................................. 28
Figure 3.4: Example of how Task Slots can be used in Flink ............... 29
Figure 3.5: Iterate operator steps in Apache Flink .................................. 30
Figure 3.6: Delta Iterate operator steps in Apache Flink ....................... 31
Figure 4.1: Master/worker architecture of Spark components ........... 34
Figure 4.2: Master/Spark’s software stack ........................................... 35
Figure 4.3: Run-time steps followed in Spark ....................................... 36
Figure 5.1: Overview of the architecture of the system ......................... 39
Figure 6.1: Memory usage of the cluster ............................................ 44
Figure 6.2: Memory usage of the master node .................................... 45
Figure 6.3: Memory usage of the worker node on average ................... 45
Figure 6.4: CPU usage of the cluster .................................................. 46
Figure 6.5: CPU usage of the master ................................................... 47
Figure 6.6: CPU usage of the worker ................................................... 47
Figure 6.7: Incoming data rate of the Cluster’s computers .................. 48
Figure 6.8: Outgoing data rate of the Cluster’s computers ................ 48
Figure 6.9: Incoming data rate of the master node ............................... 49
Figure 6.10: Outgoing data rate of the master node ............................. 49
Figure 6.11: Incoming data rate of the master node ............................. 50
Figure 6.12: Outgoing data rate of the the master node ...................... 50
Chapter 1

Introduction

The world is moving towards an era where all its surroundings are linked to a data source and everything involved in people’s life can be captured and represented digitally. Physical world is transforming into raw information in every field related to humanity. From the internet, video and telephone data to literature and from weather and geo-spatial data to stock market data and government records, everything can be digitized nowadays.

Ever growing volumes of data, coming from different sources in rapidly increasing rates, in a variety of representation and structure, constitute what is called Big Data. A trend that brings great challenges in terms of coping and handling both bounded and unbounded data, however it presents opportunities for mining substantial insight from data sets that might have been not feasible to cope with before.

Acquiring information out of large datasets can be thus, possible in many new revolutionary ways, in a huge diversity of scientific and commercial fields, bringing along great changes in our lives[1]. A subsequent need therefore, is the utilization of the proper infrastructure for these purposes.

Although new tools and frameworks emerge that promise to gracefully face the aforementioned challenges, it is not apparent how they can differ from each other and what are the criteria to consider when it comes to choosing one, for a certain task. This is the underlying purpose of this project.

1.1 Motivation and goals

The data growth and the social media explosion has changed the way data are perceived. Processing huge volumes of data might not be enough to ac-
quire meaningful insight. There exist several scenarios where data need to be processed and analyzed in flight, even before they are stored.

Applications related to fraud detection, intelligence and surveillance or algorithmic trading cannot rely on traditional "too late architectures" as they need to take action in real time. For this reason data streams, in the form of tuples, have to be continuously analyzed and transformed in memory before stored on a disk. Processing streams of data can work either by processing each tuple as a separate event, or by processing "time windows" of data in memory, or both, according to the needs of the application.

The new movement of how unbounded data can be processed and analyzed in real time, without the need of being persisted has affected the infrastructure around Big Data. This has raised the motivation behind this project, with which we aim to contribute to the Big Data community by researching and answering the following questions:

- Which are the major candidates in data stream processing?
- What are the key differences among them?
- Which is the most suitable for a given task?
- What can be their differences when applied on a computing cluster?

In particular, the project aims to provide a full overview of state of the art frameworks and solutions that can cope with stream processing tasks and provide a deep understanding of their features and properties. Moreover, besides the comparison in a theoretical level, there will be also a comparison by using some of them for a practical application, in order to gain a better understanding of their main differences when dealing with a real task.

### 1.2 Related work

Stonebraker, Çetintemel and Zdonik [18] have defined the requirements that a real-time stream processing framework should meet in order to successfully deal with stream processing applications. Any stream processing system should be capable of:

- process data in-stream, with no need for storing them
- query data in an SQL fashion
- handle stream flaws such as delays, or missing data
- provide predictable and repeatable outcomes
- combine historical with streaming data
• guarantee the integrity and availability of data
• gracefully scale out as data grow
• respond in real time

Based on this paper, several surveys such as those from Kamburugamuve [15] which evaluates streaming engines as a combination of their programming API level and their execution engine. M. de Carvalho, Roloff and O. A. Navaux [9] enlists most of the existing infrastructure along with their main characteristics and Guenter and Hesse [14] put a set of stream processing engines in terms of throughput rates and latency, having been conducted in order to present the different solutions that exist coming either from the industry or the academic world. Their goal is mostly to provide a theoretical background of the solutions studied and outline their features, similarities and drawbacks.

Finally, as far as the practical part is concerned, Yahoo engineering has conducted a detailed benchmark\(^1\). In this benchmark three of the main candidates for Yahoo’s commercial use are put in comparison based on an advertisement application, explicitly designed for this purposes. Eventually each candidate’s behaviour in terms of processing latency is measured while the throughput rates of ingesting data to the application keeps increasing.

### 1.3 Report structure

This project is divided in 7 main sections, in addition to the introduction section.

In section 2 the state of the art solutions and frameworks that were found are presented. After a small background explanation for the sake of terminology that will follow, the main criteria that the selection was based on are explained. A theoretical background for every framework under consideration is given and finally the reasons for selecting the major candidates under evaluation are analyzed.

Section 3 introduces the first of the two major candidates which is Apache Flink. Flink’s architecture and process model is examined in detail and other features such as its memory management, iterative processing and optimization are presented.

In section 4 the same process is followed for Apache Spark, diving into more detail regarding the same features that were analyzed for Flink.

The system architecture of the computing cluster and the algorithm used for the experiment are presented in section 5. The steps followed are discussed in detail as well as the technical specifications of the infrastructure used. Also the tools and methods for benchmarking and monitoring the results are introduced.

Finally section 6 is where the results of the experiment are found and commented and in section 7 a discussion about the conclusions drawn is held.
Chapter 2

State of the art solutions and frameworks

2.1 Background

Big Data is an overloaded term used by different people to represent different concepts and its popularity is steadily increasing. The term refers to collections of data that may come from different sources, in different structure (or even without any structure) and in large volumes. As a result they cannot be processed using centralized computing techniques and therefore the necessity of new techniques for storage, management and analysis of large data sets has arisen.

One of the most well known techniques is the MapReduce programming paradigm as explained by Dean and Ghemawat [10] for processing large data sets with a parallel, distributed algorithm on a cluster. Input data are split into pieces and the Map function is applied to each one of them in order to create an intermediate set of key/value pairs. Pairs with the same key are grouped together and are given as input to the Reduce function which is responsible for computing the final single value of each key and emitting a set of key/final value pairs. A typical example is that of word count as shown in figure 2.1 where the number of occurrences for each word is calculated using the MapReduce technique.

MapReduce has influenced the birth and growth of Apache Hadoop\(^1\), an open source framework written in Java for distributed storage and processing of very large data sets on computer clusters built from commodity hardware. The base Apache Hadoop framework is composed by the following modules:

Figure 2.1. Word count example following the MapReduce paradigm. The text is split in lines and Map creates key/value pairs with word as a key and number 1 as value. All the pairs are then grouped by key and the Reduce function calculates the summation of the values for each word. The final output key/value pairs with a unique word and the number of occurrences.

- Hadoop Common, which contains libraries and utilities needed by other Hadoop modules
- Hadoop Distributed File System (HDFS), a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN, a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users’ applications
- Hadoop MapReduce, an implementation of the MapReduce programming model for large scale data processing.

One of the main reasons why Hadoop became popular, is that it is a scalable infrastructure, able to handle growing volumes of data, or to improve performance for data sets of given size. This can be achieved not only by adding more resources to a single machine (scale-up), but also by adding more commodity machines to a cluster (scale-out) as claimed by Appuswamy et al. [4].

Scalability is a core principle for all of the solutions that will be presented in this chapter, along with a resource manager for scheduling the tasks of each application and managing the computing resources. Most of the solutions leverage Hadoop YARN, or other alternatives such as Apache Mesos\(^2\), which is an open-source cluster manager that was developed at the University of California, Berkeley.

2.2 Criteria and background for the selection

The growing importance of extracting real-time business insights from an ever growing amount of data resulted into the existence of numerous stream processing solutions in the market and open source software community. Therefore the problem of selecting a few such solutions for an in-depth real-life performance comparison resulted in two main goals:

1. Defining a set of requirements for real-time streaming.
2. Selecting a subset of the state of the art satisfying these requirements.

For selecting the comparison dimensions we consulted Stonebraker, Çetintemel and Zdonik [18] that describe the eight most important requirements that stream processing solutions are meant to fulfill in order to excel in handling applications in real-time. For selecting a subset of the most important solutions in the state of the art we started with surveys by Kamburugamuve [15], M. de Carvalho, Roloff and O. A. Navaux [9] and Guenter and Hesse [14].

These surveys provided a set of potential candidates that was filtered based on the requirements included in Stonebraker, Çetintemel and Zdonik[18]. As a final step this set was verified by relevant Internet articles such as [20], by the Software’s Technical Lead of TIBCO and [19] by one of the creators of Apache Flink. Each solution that was selected was also thoroughly examined not only by the aforementioned sources, but also by the information provided on its official website.

Eventually the criteria that can describe our selection filter for forming the subset of frameworks for our study can be summarized as following:

- Data process can be done on the fly, without the need for storing data
- Stream imperfections and delays can be handled by framework
- Given the same input, it is guaranteed that every framework will provide the same output
- As data flow increases, frameworks are responsible for automatically scaling out
- Response is given in real-time or near real-time

2.2.1 Open Source frameworks

Several of the selected platforms belong to the incubator of the Apache Software Foundation (Apache.org, 2016), namely Apache Flink \(^3\), a frame-

work built on a streaming dataflow engine for distributed big data analytics, Apache Samza\(^4\), a distributed stream processing framework, Apache Spark\(^5\), a general engine for large-scale data processing, Apache Storm\(^6\) and they are all open source projects as well as Esper of the EsperTech Inc. which is a component for complex event processing and event series analysis\(^7\).

### 2.2.2 Proprietary frameworks

There is also a number of proprietary solutions like SQLstream Blaze\(^8\), a suite covering the whole range of processing of data streams from data integration to analytics and visualization, IBM Stream\(^9\), a platform that facilitates the ingestion, analysis and correlation of data in data streams and TIBCO StreamBase\(^10\), a complex event processing platform for building applications that analyze and act on streaming data in real time. Although the aforementioned solutions were found as potential candidates, in the context of this study the candidate solutions were selected out of open source frameworks, that can be readily downloaded and used, and their documentation and source code is freely available.

### 2.3 Summary

Admittedly most of the free, open source, state of the art frameworks under evaluation are projects of the Apache Software Foundation, which is a community-driven, non-profit organization that freely supports the development of open source software. The Apache stream processing solutions together with Esper from EsperTech Inc constitute the main set of state of the art stream processing solutions initially considered for this study. Table 2.1 presents the main study candidates in a tabular format where each column is the name of the solution and each row is a piece of information about the corresponding solution. Some example pieces of information are historical notes, details about their capabilities, architecture, API and language support, deployability, integration with various data sources, companies using them and the size of the respective community of committers and contributors.


14
All the information was retrieved mostly from the respective official documentation of each framework and complemented by related articles [8], [19].
Table 2.1. A concentrated, background and technical overview of the state-of-the-art, open-source frameworks (Flink, Samza, Spark, Storm and Esper) that will be considered further.

<table>
<thead>
<tr>
<th>Name of the engine</th>
<th>Flink</th>
<th>Samza</th>
<th>Spark Streaming</th>
<th>Storm</th>
<th>Esper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Official committers / contributors (GitHub)</td>
<td>22 / 154</td>
<td>13 / 46</td>
<td>44 / 797</td>
<td>29 / 200</td>
<td>1* (repository changed recently)</td>
</tr>
<tr>
<td>Batch processing</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Micro-batch (Treadmill extension)</td>
<td>No (can only be combined with SQL stores)</td>
</tr>
<tr>
<td>Latest version</td>
<td>1.0.3</td>
<td>0.10</td>
<td>1.4.1</td>
<td>0.10.1</td>
<td>5.2.0</td>
</tr>
<tr>
<td>Developer</td>
<td>Apache Software Foundation</td>
<td>Apache Software Foundation</td>
<td>Apache Software Foundation</td>
<td>AMP Lab University of California, Berkeley</td>
<td>Backtype Twitter</td>
</tr>
<tr>
<td>Used by</td>
<td>ResearchGate, Swedish ICT, Bouygues Telecom</td>
<td>LinkedIn, Uber, Pivotal, Netflix</td>
<td>Spotify, GroupMe, Twitter</td>
<td>Bosch, PayPal, Ascendite</td>
<td></td>
</tr>
<tr>
<td>Computation model</td>
<td>Streaming</td>
<td>Streaming</td>
<td>Micro-batch</td>
<td>Streaming (Micro-batch with Treadmill Foundation)</td>
<td>Complex Event Processing</td>
</tr>
<tr>
<td>Deployable</td>
<td>Locally Cluster Cloud</td>
<td>Locally Cluster Cloud</td>
<td>Locally Cluster Cloud</td>
<td>Locally Cluster Cloud</td>
<td>Locally (Only enterprise products scale out)</td>
</tr>
<tr>
<td>Resource management support</td>
<td>Flink standalone, YARN</td>
<td>YARN</td>
<td>Spark standalone, YARN, Mesos</td>
<td>Nimbus, YARN, Mesos*</td>
<td></td>
</tr>
<tr>
<td>Languages support (API)</td>
<td>Scala, Java</td>
<td>Java, Scala, Python</td>
<td>Java</td>
<td>Multi-language feature (Java, C/C++, Scala, Python, Ruby, others)</td>
<td>Event Processing Language (EPL)</td>
</tr>
<tr>
<td>Integration with data sources</td>
<td>PIn // socket / replication-based, Third-party systems (Kafka, Elasticsearch, HDFS, RabbitMQ, Twitter)</td>
<td>Kafka</td>
<td>Basic (file systems, socket connections, Akka actors) Advanced (Kafka, Fluent, Kinesis, Twitter)</td>
<td>Kestrel, RabbitMQ, AMQP, Kafka, IMS, Helix, HBase, Hive, JDBCl, Redis, Solr, EventHubs</td>
<td>Fido, socket, AMQP, Spring, IMS, Relational DBs</td>
</tr>
</tbody>
</table>


2.4 Frameworks under consideration

2.4.1 Apache Flink

Apache Flink\(^{11}\) is a platform for distributed stream and large-scale batch data processing. It is a continuation of the Stratosphere project\(^{12}\), that started as a collaboration between TU Berlin, Humboldt University and the Hasso Plattner Institute in 2010. Eventually it became an Apache Incubator project in March 2014. Currently it is one of Apache’s top projects, having 22 active committers and more than 150 contributors in the Github account. There are several organizations, institutes and companies that have included the usage of Flink, among them Bouygues Telecom, ResearchGate and SICS Swedish ICT. A more detailed list conducted by one of the creators of Flink, Kostas Tzoumas, can be found in the Flink’s section in Apache’s Foundation wiki\(^{13}\).

Flink can be deployed locally in a single Java Virtual Machine(JVM) for testing purposes, in the cloud, or in a cluster consisting of numerous nodes where resources can be handled by its standalone manager, or by Yet Another Resource Negotiator (YARN)\(^{14}\). Flink’s core is built on a data flow streaming engine, responsible for executing a program received in the form of a job graph, which defines the number of tasks and the way the tasks consume and produce data. One of the fundamental functionalities of this engine is pipelining, meaning that all tasks must be online simultaneously in order for the data to be able to flow through the tasks regardless of the node, that a task might reside on. An example of how Flink’s engine works in a simple case is shown in figure 2.2.

A job graph is provided to Flink by the use of two APIs: the DataSet API and DataStream API that are based on the DataSet (the main abstraction for batch processing) and the DataStream (the main abstraction for stream processing), respectively. DataStream is responsible for building the stream that data have to follow, according to the plan that is created. However Flink also supports batch processing by considering batches as finite streams of data. Thus, the DataSet API not only builds the job graph in that case, but also uses an optimizer to create the optimal plan. These two APIs can be exploited by developers in Java or Scala and there are also a few other extensions and libraries, either as complete features, or as potential components. At the time that this study is conducted, Flink 0.10.1 comes with:

Figure 2.2. Flink’s workflow. The input DataStream, consisting of words, is split in tuples. Every tuple flows through a task that resides on a node of the cluster. The particular task here is a Map transformation where every word is assigned number 1. The output that is emitted is another tuples DataStream

- a Beta version of FlinkML, a machine learning library that supports several algorithms such as supervised learning, data preprocessing, and recommendations
- Gelly which is a Graph API for simplifying the development of graph analysis applications in Flink and last but not least
- the Table API, an API for performing SQL-like operations in either batch or streaming applications, instead of manipulating DataStream or DataSet abstractions

2.4.2 Apache Samza

Apache Samza\textsuperscript{15} is a distributed stream processing framework that was originally developed by Linkedin\textsuperscript{16} in order to process data in real time and overcome the latency problem that existed when using Hadoop\textsuperscript{17} for batch data processing. Eventually it was open sourced in 2013 when it also became a member of Apache’s incubator. Currently it belongs to Apache’s top projects with 13 committers and 46 contributors in the Github repository\textsuperscript{18}. According to the list of Samza users\textsuperscript{19} officially posted in Apache’s wikipage apart

\textsuperscript{15}http://samza.apache.org [Accessed 14 Mar. 2016]
\textsuperscript{17}http://hadoop.apache.org [Accessed 14 Mar. 2016]
\textsuperscript{18}https://github.com/apache/samza [Accessed 14 Mar. 2016]
\textsuperscript{19}https://cwiki.apache.org/confluence/display/SAMZA/Powered+By [Accessed 14 Mar. 2016]
from LinkedIn there are several other companies that leverage the potential of Samza such as Netflix (Netflix.com, 2016) and Uber (Uber.com, 2016).

In terms of its architecture, Samza is made up of three layers (streaming, execution, and processing) and follows a similar approach to Hadoop. Samza and Hadoop both accommodate YARN as the resource manager to schedule the jobs across numerous nodes, in the execution layer. However they differ in terms of the source of data ingestion. Hadoop stores and retrieves data from the Hadoop Distributed File System (HDFS) while Samza integrates Kafka\textsuperscript{20} in the streaming layer. Kafka is a message queuing system which is able to guarantee that no message is going to be lost. Finally instead of the MapReduce paradigm followed by Hadoop, Samza provides its own API in Java for the processing layer. Although YARN and Kafka (along with the Samza API) are put together to form what is called Apache Samza, the actual framework is not limited to these two. The execution and streaming layers are pluggable and developers are able to replace them with alternatives.

The two principal high level abstractions in Samza are Streams and Jobs. A Stream is a set of immutable messages that belong to the same category, or are of similar type. For instance these can be the logs produced by a service, or any kind of event data. Streams are broken into one or more Partitions, or in other words, sequences of ordered messages within the Streams. In turn, Streams are consumed or produced by Jobs which actually represents the logical transformations applied to the certain Stream. Similarly to Streams, Jobs are broken into Tasks which are single units that allow parallelization. Each Task consumes data from one Partition, for each of the Job’s input Streams. An overview of how the aforementioned abstractions interact is shown below in Figure 2.3 adapted from Samza’s official documentation\textsuperscript{21}

2.4.3 Apache Spark

Apache Spark\textsuperscript{22} is a general-purpose engine for data-processing in large scale. It was originally developed in the Algorithms-Machines-People Lab (AMPLab) at California, Berkeley and was later donated to the Apache Software Foundation. At the time of this study, Spark is among the top projects of Apache and has one of the biggest communities in the field with 44 committers and approximately 800 contributors in Github. That said, Spark is claimed to be one of the most popular\textsuperscript{23} frameworks for large-scale data processing.

\textsuperscript{21}https://samza.apache.org/learn/documentation/0.10/introduction/concepts.html [Accessed 11 May 2016]
\textsuperscript{22}http://spark.apache.org [Accessed 14 Mar. 2016]
Figure 2.3. Samza’s workflow. The input Stream, consisting of words, is broken in two Partitions, p0 and p1. A Map Job is parallelized to several Tasks (as many as the number of partitions) and each one of them is assigned by YARN to a machine, so as to be independent. Eventually an output Stream is produced.

A long list\textsuperscript{24} exists in Apache wikipedia\textsuperscript{25}, with companies such as Groupon (Groupon.com, 2016), Yahoo! (Yahoo.com, 2016) and eBay Inc (Ebay.com, 2016). leveraging Spark’s capabilities in various ways and companies such as Guavus (Guavus.com, 2016), Kelkoo (English Kelkoo.com B2B, 2016) and Opentable (Opentable.com, 2016) using Spark Streaming in particular.

The main abstraction that Spark provides is a Resilient Distributed Dataset (RDD) \cite{2}, which is a collection of elements partitioned across the nodes of a cluster that can be operated on in parallel. When it comes to Spark Streaming, which is Spark’s extension that enables data stream processing, the high-level abstraction is called Discretized Streams (DStreams) \cite{17} and it represents a continuous stream of data that are either ingested, or produced, to or from Spark’s engine. What is noteworthy is that a DStream internally is represented by several RDDs and each one contains data from a certain time interval. Thus every operation is practically being done on RDDs, leading to a micro-batch approach, where essentially every stream is considered as an infinite number of micro batches. Spark’s workflow is illustrated in figure 2.4 below.

Spark Streaming is able to ingest data from both basic (e.g. file systems, sockets) and advanced sources (e.g. Twitter (Twitter.com, 2016), Flume\textsuperscript{26}, Kafka\textsuperscript{27}, can be deployed locally for testing purposes, in a cluster (with Standalone, Mesos\textsuperscript{28}, or YARN resource management), or in the cloud, offers

\textsuperscript{24}https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark [Accessed 14 Mar. 2016]
\textsuperscript{26}https://flume.apache.org [Accessed 10 May 2016]
\textsuperscript{27}http://kafka.apache.org [Accessed 10 May 2016]
\textsuperscript{28}http://mesos.apache.org [Accessed 10 May 2016]
complete APIs in Scala, Java and currently a limited one in Python and finally it can also exploit Spark’s ML library for applying several type of either on-line machine learning algorithms, or compare streams of data in real time with models that were created of historical data.

Figure 2.4. Spark’s workflow. An input stream of words is divided to micro RDDs according to the time interval they were ingested. These RDDs constitute the input DStream. A map transformation is applied to each one and the final output is another DStream that consists of micro RDDs respectively.

2.4.4 Apache Storm

Apache Storm\textsuperscript{29} is a distributed real-time computational framework that was originally released in 2011 by Backtype and was later acquired by Twitter and got open-sourced under the Apache Licence. Currently Storm is one of the top Apache projects, officially supported by 29 committers while 200 people have contributed in its Github repository overall. In its official website, the companies that use Storm are enlisted in detail\textsuperscript{30}, including among others, Spotify (Spotify.com, 2016) and Twitter (Twitter.com, 2016).

Any Storm application for real-time manipulation of data is described by a concept called Topology, which is similar enough to a MapReduce task in Hadoop. Nevertheless their main difference is that a MapReduce task will eventually terminate, whereas a Topology will run forever (unless it is terminated by the user). Topology is actually a graph that consists of two types of components, Spouts and Bolts. The main abstraction of Storm, Stream, which is an unbound series of tuples of data, can be emitted to the topology through Spouts. All the transformations applied to a Stream afterwards, are

\textsuperscript{29}http://storm.apache.org [Accessed 14 Mar. 2016]
\textsuperscript{30}http://storm.apache.org/documentation/Powered-By.html [Accessed 14 Mar. 2016]
represented by Bolts, which can vary in complexity, from simple transformations like filtering, to more sophisticated functions. Therefore it is likely that multiple Bolts have to be applied to a single Stream. A simple Storm topology for mapping words with number 1 is shown below.

![Storm's workflow diagram](image)

*Figure 2.5. Storm’s workflow. The Topology includes two Spouts, one that emits word tuples and another that emits line tuples. A Bolt is applied first to the lines Stream to split them in words and emit word tuples and another Bolt is applied to word tuples to create pairs of the format <word,1>*

Storm can be deployed either in a cluster, or in local mode which is a simulation of a Storm cluster locally in a single machine and can prove useful for developing and testing topologies. Although there is support for the YARN resource manager, Storm originally uses a daemon called Nimbus, for managing the resources of a Storm cluster and distributing work across its nodes.

Bolts and Spouts can be written in Java, however Storm was originally designed in a way to support any language. There exist adapters that also implement communication with Bolts and Spouts written in Python, Javascript, Perl and Ruby. Finally, Storm contains modules for ingesting data from several types of sources. Typical examples are Kafka (Kafka.apache.org, 2016), HDFS, HBase (Hbase.apache.org, 2016), Hive (Hive.apache.org, 2016), Java Database Connectivity (JDBC), Solr (Lucene.apache.org, 2016), Redis ((Redis.io, 2016)) and Azure EventHubs (Azure.microsoft.com, 2016).
2.4.5 Esper

Esper\textsuperscript{31} is an open source event series analysis and Complex Event Processing (CEP) engine that has been released since 2006 and is further supported and extended commercially by EsperTech Inc. Since it is a relatively old project, its source code had been developed initially in Codehaus and recently was transferred to GitHub, however at the time that this study is conducted, detailed information regarding the number of contributors and committers could not be retrieved from reliable sources. A list of EsperTech Inc. customers, including Bosch (www.bosch.com), PayPal (www.paypal.com) and Accenture (www.accenture.com), can be found in the official website’s partners section\textsuperscript{32}.

Slightly different than the aforementioned solutions, Esper is more of a component, rather than a complete Extract-Transform-Load (ETL) framework, that focuses more on high speed querying of data streams or events, recognizing patterns and analyzing data in real-time. In order to achieve those, Esper comes with its own SQL-like language called Event Processing Language (EPL). It is available as Esper and NEesper which are embeddable components available to Java-based and .NET-based processes respectively.

Esper provides input adapters for several stream sources, such as files (e.g. .csv), Java Message Services (JMS), relational databases, HTTP and sockets. However a certain drawback in the open source distribution of Esper, is that EsperTech Inc. only considers scaling with respect to throughput from a few tens of thousands messages per second up to several hundreds of thousands per second and scaling up by adding CPU and memory to the infrastructure. Thus there is no consideration for scaling out across several JVMs or systems, what is a feature goal of the company’s other products, namely the EsperHA (HA stands for High Availability) and the Enterprise Edition. The only way to achieve scaling out by using Esper is to include it as a component of one of the previously mentioned frameworks, e.g. in Apache Storm, embedding Esper in a Java Bolt.

2.5 Selection of major candidates under evaluation

Based on what was mentioned previously, several reasons led us to select Apache Spark streaming and Apache Flink for a more in depth comparison both from an architecture perspective as well as from a practical application.

On the one hand, Apache Spark is the most active project (in terms of community numbers) encountered, utilizing the micro-batch approach which

---


is slightly different from what the rest of the frameworks do for processing streams of data. Moreover Spark supports batch processing, being a general purpose framework, allowing an easy transition from batch processing to stream processing applications and vice versa, or facilitating the combination of both features in a single application.

On the other hand Flink is the only framework that can purely support the same capabilities (by translating big batches of data to finite streams), while having a completely different engine under the hood. Its data flow engine gives Flink a more streaming-oriented identity, while it arises a challenge for Spark in terms of handling and processing data streams.

Finally they both offer similar APIs that are based on and extend the MapReduce paradigm. Additionally it is possible to write an application in a language that both support (either Scala or Java), and test it first in the same local instance and later in the same computing cluster, in order to study the scalability potentials that the two frameworks can support.
Chapter 3

Apache Flink

3.1 General architecture and process model

Apache Flink is the first of the major candidates for this study. As formerly mentioned it is a platform for distributed stream and large-scale batch data processing drawing its inspiration from dataflow systems (e.g. Google Data Flow [2]). Flink is built on a native stream engine, where each record of data flows through long standing tasks in a pipelined manner and therefore data can be accessed and processed in time. A Flink streaming application is not executed directly when it is submitted. On the contrary, when the Flink system is started, it first brings up a JobManager (master) and one or more TaskManagers (workers). If the system is started in local mode, then both the JobManager and the TaskManager will be brought up in the same Java Virtual Machine (JVM).

Subsequently, a client is created which, by using the DataStream API, translates the program to a JobGraph, or in other words the plan that has to be executed which is represented by a graph of parallel operators and data streams between those operators. This execution plan will be handled by the JobManager, who is responsible for scheduling and distributing the tasks across the TaskManagers, monitoring each worker’s progress, deploying tasks to existing or new workers and stopping/canceling tasks running on current workers.

The exact process followed by Flink is depicted in Figure 3.1 adapted from Flink’s documentation regarding its general architecture and process model \(^1\).

Seen from another perspective Flink consists of three major components that are built on top of each other:

\(^1\)https://ci.apache.org/projects/flink/flink-docs-release-1.0/internals/general_arch.html [Accessed 11 May 2016]
The runtime layer receives the program as a JobGraph. A JobGraph is a way to represent the data flow with tasks whose inputs and outputs are data streams.

The DataStream API and the DataSet API are responsible for generating JobGraphs following different processes. The DataSet API exploits an optimizer for selecting the optimal plan according to which data will flow, while the DataStream API uses a stream builder.

The deployment layer which defines how the JobGraph will be executed (e.g. locally, remotely, YARN, etc.)

Finally there are Libraries and APIs that are bundled with Flink and generate DataSet or DataStream API programs. These are: a) Table for queries on logical tables, b) FlinkML for Machine Learning, c) Gelly for graph processing and d) CEP for event processing. The full software stack of Flink is shown in Figure 3.2 as adapted from Flink’s documentation for architecture and process model².

²https://ci.apache.org/projects/flink/flink-docs-release-1.0/internal/general_arch.html
3.2 Jobs and scheduling

3.2.1 Job manager data structures

When a job is executed in Flink, the JobManager is responsible for distributing, monitoring and scheduling tasks as well as handling finished tasks or reacting to any kind of failures. Initially the JobManager receives the JobGraph (generated by either the DataStream API or DataSet API) which is an overall representation of how the data will flow through various operators (JobVertex) and what will the intermediate results (IntermediateDataSet) be.

As depicted in Figure 3.3 (taken from Flink’s documentation about JobManager’s Data Structures) the JobManager converts the JobGraph into an ExecutionGraph which is practically a more detailed version of JobGraph, containing more information regarding the parallelism of each JobVertex and the IntermediateDataSet partitioning. Therefore the eventual ExecutionGraph contains a number of parallel subtasks (ExecutionVertex) for each JobVertex and a number of partitions (IntermediateResultPartition) for each intermediate result.

3.2.2 Scheduling

The actual resources for executing tasks in Flink are called Task Slots and are held by TaskManager(s). Each Task Slot is capable of running a pipeline of multiple consecutive parallel tasks.

Figure 3.3. Flink’s JobManager conversion from JobGraph to ExecutionGraph

For instance, Figure 3.4 as found in Flink’s official website\(^4\) illustrates how the resources (Task Slots) are going to be exploited when a data source and a map function with a parallelism of 4 and a reduce function with a parallelism of 3 are assigned to two TaskManagers (TaskManager 1, TaskManager 2) with 3 slots each.

3.3 Iterative processing

Data processing and analysis is not merely limited to plain transformations such as filtering data, or basic map-reduce tasks. As the need of getting deeper insight out of our data increases, the application of machine learning or graph processing algorithms might become crucial. In both cases these algorithms are characterized by the occurrence of iterative algorithms from clustering and gradient descent to Page-Rank and path algorithms on graphs. For this reason Flink provides a special iteration operator influenced by project Stratosphere which embeds a function known as the step function, as explained by Ewen et. al [11]. There are two variants of this iteration operator:

(a) Iterate operator
(b) Delta Iterate operator

\(^4\)https://ci.apache.org/projects/flink/flink-docs-release-1.0/internals/job_scheduling.html
Both operators repeatedly invoke the step function on the current iteration state until a certain termination condition is reached.

3.3.1 Iterate operator

The iterate operator is used for simpler cases of iterations: in each iteration, the whole input (it can either be the initial data or previous results) is ingested, and the next step of the partial solution (e.g. map, reduce, join, etc.) is produced.

1. Iteration Input: Initial input for the first iteration from a data source or previous operators.

2. Step Function: The function that will be invoked in every iteration. It is an arbitrary data flow consisting of operators like map, reduce, join, etc. defined by the particular task to be executed.

3. Next Partial Solution: The output of the step function after every iteration is given back as an input.

4. Iteration Result: The output of the last iteration is written to a data sink or used as input to the following operators.

The aforementioned steps are depicted in Figure 3.5, as found in Flink’s official documentation\(^5\). As far as the termination conditions for an iteration, there can be multiple options:

\(^5\)https://ci.apache.org/projects/flink/flink-docs-release-0.7/iterations.html

29
Figure 3.5. Iterate operator steps in Apache Flink

(a) Maximum number of iterations: Without any further conditions, the iteration will be executed this many times.

(b) Custom aggregator convergence: Iterations allow to specify custom aggregators and convergence criteria like sum aggregate the number of emitted records (aggregator) and terminate if this number is zero (convergence criterion).

Maximum number of iterations: Without any further conditions, the iteration will be executed this many times. Custom aggregator convergence: Iterations allow to specify custom aggregators and convergence criteria like sum aggregate the number of emitted records (aggregator) and terminate if this number is zero (convergence criterion).

3.3.2 Delta Iterate operator

The delta iterate operator is used when dealing with the more sophisticated scenario of incremental iterations. In incremental iterations, instead of recomputing the solution in each iteration, only a subset of elements of the solution are altered and evolve to the final solution. Community analysis or connected component computations in graphs, are some of the examples where an algorithm can be significantly improved with the use of Delta Iterate operators.

Therefore not every element of the initial data set is necessarily modified in every iteration. Subsequently this results to improved algorithms. The most complex parts of the algorithm are first dealt with, which causes iterations in later stages to affect a small subset of the initial data set.

1. Iteration Input: The input for the first iteration includes the initial workset and the solution set. They can either come from data sources or previous operators.

2. Step Function: The function that will be invoked in every iteration. It is an arbitrary data flow consisting of operators like map, reduce, join, etc. defined by the particular task to be executed.
3. Next Workset/Update Solution Set: The iterative computation is leveraged by the next workset, which will be ingested to the next iteration. Moreover, the solution set will be updated without the need to be reconstructed and will be forwarded. Different operators of the step function can modify the two data sets.

4. Iteration Result: The output of the last iteration is written to a data sink or used as input to the following operators.

![Figure 3.6. Delta Iterate operator steps in Apache Flink](image)

The way Delta Iterate operators are organized is illustrated in Figure 3.6, as found in Flink’s official documentation\(^5\). By default, the delta iterator will terminate either when the next workset that is calculated is empty, or when we have reached a maximum number of iterations.

3.4 Memory management

Memory is considered as a set of Memory Segments by Flink. Every segment serves as a part of memory (by default is 32KB) and it constitutes the basic unit of memory backed by a Java byte array. When a record is stored in Flink, it is in fact being serialized into one or more memory segments. Additionally, a "pointer" for this record may be added in another data structure. This allows Flink to not only efficiently serialize records but also breaking them across pages.

\(^5\)https://ci.apache.org/projects/flink/flink-docs-release-0.7/iterations.html
3.4.1 Managed memory

The actual way that Flink splits the JVM Heap is explained below. Flink divides the JVM Heap in the following three regions:

(a) Network buffers: A set of 32 KB buffers (that can be configured from `taskmanager.network.bufferSizeInBytes`) used to buffer records before transmitting over the network.

(b) Memory Manager pool: A large number of 32 KB buffers used by runtime algorithms in case they need to buffer records. Every record will be stored in serialized form in these segments as described previously.

(c) Remaining (Free) Heap: This is the part of the heap that will remain mostly for the user code and any of the Taskmanager’s data structures, which are usually small however.

3.4.2 Garbage collector

The aforementioned memory management mechanism has an important impact on the garbage collection behavior of Flink. Records are not gathered as objects, instead they are serialized as memory segments which are the only long lived objects. However they can be reused for different records and are never being garbage collected. Thus, the only case of long lived records are those that pass through user functions to be serialized into memory segments.

3.5 Optimization

Apache Flink includes a cost-based optimizer which is independent of the actual programming interface, introduced by project Stratosphere [3]. This optimizer’s task is to estimate the execution cost of different plans, using different strategies and eventually select the one with the smallest estimated cost.
Chapter 4

Apache Spark

4.1 General architecture and process model

Apache Spark is the second of the major candidates for this study. It is another distributed, general purpose framework, being able to process data coming from both batch and streaming applications in large scale. As opposed to Flink, the stream data processing is not technically based on a purely stream engine, rather than on a micro-batch approach. Spark Streaming extension of Spark’s platform introduces DStreams as a new abstraction (which consists of RDDs, Spark’s main abstraction). DStreams are practically continuous sequences of RDDs and therefore all the actions and transformations applied to a DStream, are eventually translated to actions and transformations on the underlying RDDs. That said it is apparent that Spark is eventually exploiting the same abstraction (RDD) either for batch or stream processing. As a result this is a fact which enables an easy transition from a batch processing task to a stream processing one and vice versa, while it is also valuable when it comes to co-existence of both types of processing in the same program.

Spark follows a master/worker approach which is described in [16]. The central coordinator is called the driver which communicates with one or more workers which are called executors. The driver and its executors are independent Java processes that together consist a Spark application. Every Spark application has to initially create an object called SparkContext which practically coordinates the driver with the executors and connects to the respective Cluster Manager (Mesos, YARN, Standalone) for allocating the required resources for the application. The communication of the aforementioned components is illustrated in Figure 4.1 taken from Spark’s documentation regarding Cluster Components 1.

1http://spark.apache.org/docs/latest/cluster-overview.html
Finally, apart from pure batch processing and Spark Streaming that is the focus of this study, Spark provides few more features and extensions that cover a wide range of options when it comes to data analysis. Therefore, besides Spark’s core which is directly connected with the RDD abstraction and the capability of connecting to different resource managers, Spark’s software stack is complemented with components such as Spark SQL that allows querying structured data within Spark programs, MLlib, Spark’s scalable machine learning library and GraphX, an API for graphs and graph-parallel computation. An overview of Spark’s software stack is shown in Figure 4.2. It is noteworthy that both frameworks offer capabilities that cover a range of similar needs apart from batch or stream processing. Querying data in SQL manner (Spark SQL and Table), applying machine learning algorithms to datasets (MLlib and FlinkML) and graph processing (Gelly and GraphX) can be supported by both frameworks. Therefore, although outside of the scope of this study, the aforementioned directions could be further discussed and put the two frameworks under comparison.

4.2 Jobs and scheduling

In order for an application to access the Spark Streaming library, an instance of a class called StreamingContext has to be created and initialized. StreamingContext (which is found in driver program and is the equivalent of SparkContext for batch Spark applications) is the main entry point for all streaming functionality as described in [13]. In order to be initialized the information that StreamingContext requires as arguments are the URL of the computing cluster that will host the application (this can also be the local-
host when an application is still being developed or tested) and the size of the micro-batches, the DStream is going to be split to.

A Spark program implicitly creates a Direct Acyclic Graph (DAG) \[^{16}\] which is a graph that includes the stages of executions within the program. The nodes of this graph are RDD partitions (smaller parts of RDDs) while the edges are transformations being applied on top of these partitions. Spark is written in Scala where objects are immutable and this is also transferred to Spark where instead of updating the old ones (Acyclic), new partitions are created after each transformation by transferring data directly from older partitions (Direct). This graph will be further split to Tasks (which are the smallest unit of work in Spark) and Stages which consist of the tasks applied to partitions. The driver which acknowledges every information available about the existing workers will then execute the task scheduling, where all tasks are assigned to the respective executors via the Cluster Manager (Mesos/YARN/Standalone). Finally the executors will have two roles. First to hold the threads that will execute the tasks they were assigned with and second to provide in-memory storage for RDDs so they can be faster revoked when needed.

Eventually when the driver program runs, the DAG will be first converted into a physical execution plan. The steps followed in run-time are illustrated in detail in Figure 4.3.

### 4.3 Iterative processing

As previously mentioned for Apache Flink, data analysis has gone beyond the boundaries of simple tasks like filtering or aggregation functions on data streams. Two of the most substantial components of the Apache Spark soft-
ware stack are its machine learning library (MLlib) and GraphX, for graph computations. That said, Spark is widely used for providing more meaningful insight in big data sets, by facilitating the application of machine learning and graph processing algorithms.

That is achieved thanks to the way data are distributed and reliably stored in-memory with Spark’s core abstraction (RDD). Users are allowed to explicitly cache a dataset with the .cache() operation. This means that data will be accessed in RAM memory instead of the disk.

Storing data in-memory is therefore the key that allows the avoidance of unnecessary and costly disk accesses and dramatically improves the application of iterative algorithms efficient by having subsequent iterations share data through memory, or repeatedly accessing the same dataset.

4.4 Memory management

Memory in Spark can be divided in two main categories, depending on the purpose it is being used for.

(a) Execution memory, which refers to the part of memory being used for computation shuffles, joins, sorts, and aggregations

(b) Storage memory, which refers to the part of memory being used for caching and propagating internal data across the cluster.

Execution and Storage memory share a unified memory region (M), making Spark flexible in how memory is used, depending on the type of application. When no execution memory is used storage can acquire all the available memory and vice versa. Execution memory can evict storage if needed, until a certain threshold (R) meaning that R will be a subregion where cached blocks are
never evicted. However storage may not evict execution due to complexities in implementation.

This design of Spark makes it quite flexible in terms of memory usage, depending on the nature of the application. In particular applications that do not use, or need caching of RDDs in order to enhance reusability of data and avoid overhead of recomputation can use the entire memory space for execution. On the other hand applications that need to use caching and persisting of RDDs can ensure that a minimum storage space will be reserved without requiring the user to know of how memory is divided internally.

Although memory is not managed explicitly in Spark, as applications aim towards better performances, the overhead of JVM objects and Garbage Collector can become quite considerable. In such cases Spark can offer various ways of tuning, or reducing memory consumption.

4.4.1 Data structures

As far as the data structures used in Spark applications, memory consumption can be reduced when avoiding Java features such as pointer-based data structures and wrapper objects. This can be done in several ways such as

(a) Designing data structures to consist of arrays of objects and/or primitive types, instead of standard Java or Scala collection classes (i.e. HashMap)

(b) Avoiding nested structures with a lot of small objects and pointers

(c) Using numeric IDs or enumeration objects instead of strings for keys

4.4.2 Serialization

Another way to reduce memory consumption when the objects are still too large, despite the Data Structures tuning, is to store them in serialized form. Spark will then store each RDD partition as one large byte array. The only downside of storing data in serialized form is slower access time when using the default Java serialization. However, Spark can also use the Kryo library, which is significantly faster and more compact than Java serialization.

4.4.3 Garbage collector

JVM garbage collector may become a problem especially when RDDs that are stored in a program need to be reused. When Java needs to evict old unused objects to make room for new ones, it will have to go through all of the Java
objects that exist. That said the cost of the garbage collector is proportional to
the number of Java objects, so using data structures with fewer objects can sig-
nificantly reduce the cost. Moreover persisting objects in serialized form, will
result in one object (potentially a large byte array as described previously) per
RDD partition which can also contribute in lowering the cost of the garbage
collector.

4.4.4 Project Tungsten

Databricks has recently announced Project Tungsten\(^2\), which is claimed to
be the largest change Spark has witnessed since its inception. A core initiative
of Project Tungsten is improving efficiency of memory for Spark application
by leveraging application semantics to manage memory explicitly and elimi-
nate the overhead of JVM object model and garbage collection.

Databricks has acknowledged the two major problems being the inherent
memory overhead of Java objects and the overhead and unreliability of the
garbage collector when it comes to successfully estimating lifecycle of the
objects. That said the aspiration is to tackle both of those problems by intro-
ducing an explicit memory manager to convert most Spark operations directly
against binary data rather than Java objects.

4.5 Optimization

Apache Spark does not come with a built-in automatic optimization when
it comes to data stream processing and therefore Spark jobs need to be man-
ually optimized and adapted to specific data streams. That said the user has
to acquire a deep knowledge of how Spark works internally in order to pick
or avoid operators accordingly to result in having an efficient and well tuned
Spark application.

The only component that uses an optimizer is Spark SQL that allows rela-
tional data processing in Spark [5]. Spark SQL uses the Catalyst optimizer in
particular, which supports both rule-based and cost-based optimization. How-
ever this component is beyond the scope of this study.

\(^2\)https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-
metal.html
Chapter 5

System architecture and experimentation methodology

In this chapter the methodology that was followed for the experiment to take place is described. We compare the two processing engines, Flink and Spark, in a cluster environment with a given benchmarking algorithm. An attempt to express this algorithm using PMML for creating a common, interoperable model failed. The reasons are explained in section 5.4. Nevertheless, both frameworks were installed in the cluster described in section 5.1, developed the common benchmark (described in section 5.2) and run the experiments following the methodology that is presented in section 5.3. An overview of the whole procedure is illustrated in Figure 5.1.

![Figure 5.1. Overview of the architecture of the system](image-url)
5.1 Computing cluster

5.1.1 Nodes

Our computing cluster consists of six machines used as workers and one used as the master node. Below there is a representation of the architecture and the machines specifications. All the machines are running Linux 3.13.0-62-generic (x86_64) as operating system, in particular Ubuntu 14.04.3 LTS.

Master / Driver

    Master node CPU: 1 x 2.50GHz RAM: 9.77GB

Workers

    data node 1 CPU: 4 x 2.50GHz RAM: 15.67GB
    data node 2 CPU: 4 x 2.50GHz RAM: 15.67GB
    data node 3 CPU: 4 x 2.50GHz RAM: 15.67GB
    data node 4 CPU: 4 x 1.90GHz RAM: 15.67GB
    data node 5 CPU: 4 x 1.90GHz RAM: 15.67GB
    data node 6 CPU: 4 x 1.90GHz RAM: 15.67GB

The versions of Flink and Spark installed on the cluster are 0.10.0 and 1.4.1 respectively. Flink in particular allows starting the cluster in a streaming mode (apart from the normal one) that is optimized when running only streaming jobs. The streaming mode changes the startup behavior of Flink. The system is not bringing up the managed memory services with preallocated memory at the beginning. Flink streaming is not using the managed memory employed by the batch operators. By not starting these services with preallocated memory, streaming jobs can benefit from more heap space being available.

5.1.2 Network characteristics

The overall cluster performance depends on the cluster node resources such as number of CPUs, CPU speed, and RAM, and the characteristics of the network connecting the cluster nodes. In order to measure the characteristics of the links between the cluster nodes, we used iperf. Iperf is a reliable network tool that is commonly used for testing network or cluster performance by creating Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams and measuring the throughput of the network for those data streams.

First an iperf server was started in one of the cluster's machines by manually invoking

    iperf -server

and then on another machine an iperf client was invoked
iperf -client <server> -time 60 -interval 5 -parallel 1
dualtest

The test was run for one minute for every master-worker connection. The average bandwidth from each worker to the master and vice versa can seen in Table 5.1.

Table 5.1. Network measurements from master node to worker nodes

<table>
<thead>
<tr>
<th>Node</th>
<th>Avg bandwidth in Gbits/s (worker to master)</th>
<th>Avg bandwidth in Gbits/s (master to worker)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.01</td>
<td>3.77</td>
</tr>
<tr>
<td>2</td>
<td>5.50</td>
<td>6.81</td>
</tr>
<tr>
<td>3</td>
<td>6.63</td>
<td>5.86</td>
</tr>
<tr>
<td>4</td>
<td>3.70</td>
<td>3.47</td>
</tr>
<tr>
<td>5</td>
<td>6.01</td>
<td>6.86</td>
</tr>
<tr>
<td>6</td>
<td>6.47</td>
<td>6.65</td>
</tr>
</tbody>
</table>

Depending on the results we will get when running the experiment, this can give us an estimation about whether the network capacity in our cluster can become a bottleneck, affecting the application or not.

5.2 Application

The application that will be used in order to evaluate how the different frameworks perform is described through a telemetry data monitoring system for the city of Stockholm as follows.

Every second we receive a number of events in a string format from different users in the form of <user_id, longitude, latitude, timestamp>, which represents a certain user’s location at a given moment. This information could originate from different people’s smartphones, but for the sake of this experiment we have used a data set that already contains a big number of records of this format. Each record has an approximate size of 50 Bytes. Moreover, the user_id values are not related with any real user identity whatsoever but are rather arbitrary identifiers. Every event is processed and a single Passenger scala case class instance is built which holds the aforementioned information (location, time) for every user, every second.

After the pre-processing and filtering of the incoming data, the application uses time-based sliding time windows and applies aggregation functions to identify whether a user is moving or not within a certain timespan from a Box to another. If so, the Passenger instance is changed to MovingPassenger and by accumulating MovingPassenger instances from different users, we finally try to figure out trips (meaning a route from one location to another at a certain time) that could be common among our users.

These trips are represented by a case class called ActiveTrip which is eventually used to get an overview called as TripSummary, of how users (potential passengers)
move in terms of location within a city during certain time slots. Finally the number of occurrences of each TripSummary is calculated and printed.

The aforementioned system could be very helpful to acquire valuable insight of how people move within the limits of any geographical area. As a result there can be several important observations regarding popular destinations in different times of the day and routes that are more followed than others. People that passively provide these data either when moving by car, or on foot could be seen as potential passengers of transportation means that can be found in this geographical area.

When it comes to public transportation. Having this information in real-time can help authorities involved in public transportation, with monitoring and following the trends and how they change at any given timeslot. Subsequently this application could be the prototype for a mobility monitoring system, based on which timetables for buses, trains, or other transportation means can be adjusted accordingly.

5.3 Methodology and monitoring

For the sake of this experiment a data file taken from Kaggle\(^1\) with 357434921 tuples was used. We have created a script that replays the data from the data file and generates streams of data in the very same format of our application input. In both Spark and Flink, streams are received over a TCP socket and data are redistributed to all the nodes that are available in the cluster. This way we have full control of the rate with which data are ingested through our application.

Since our application leverages window operators we had to adjust the code for Spark so that the sliding window interval is multiple of the batch interval we have already selected to ingest data in our application. In order to have an equal comparison we have chosen equal values for the sliding windows and the micro-batches to be aggregated every 2 seconds, having a window size of 30 seconds.

We set the throughput to be 10000 events per second and begin with the case where there is only one worker node along with the master node. Then, after we monitor the behavior of the cluster and this unique worker node in particular, we gradually increase the number of worker nodes, (while still maintaining the same throughput rate) to observe how the application is going to scale out in terms of CPU percentage, memory usage and network transmission rates between the master and the worker nodes. The basic settings for each worker node (computer) in the cluster is 2048 MBs of RAM memory and a parallelism of 4, meaning that each computer’s core can handle one task at a time since all the worker nodes have 4 cores each. The basic settings for the driver/master node in both cases allocates the same amount of memory (2048 MBs) as the worker nodes.

For the needs of monitoring the utilization of the cluster resources by the different frameworks the Ganglia Monitoring System \(^2\) was used. Ganglia was developed and

\(^1\)https://www.kaggle.com/

\(^2\)http://ganglia.info/
open-sourced by the University of California, Berkeley and is a scalable distributed monitoring system for high-performance computing systems such as our cluster. Ganglia provides several metrics and measurements such as CPU utilization, memory usage and data sent and received through our network. All the metrics are visualized both for the cluster as a whole and for each node explicitly. More information about the features and capabilities of Ganglia can be found in the official website and book [17].

5.4 PMML - a standardization attempt

The goal of this project was to compare and evaluate how different stream processing engines cope with applying transformations and actions on continuous flows of data. That said, in order to guarantee that the procedure followed for evaluating each framework is precisely the same in every case, it would be of essential importance to develop a model in a common language as a standard. This would result not only to an easier way to provide the same processing for every engine but in the same time it could facilitate the process of plugging in more frameworks to undergo the procedure.

For this reason PMML, Predictive Model Markup Language) [12] was taken under consideration. PMML is an open standard developed by the DMG (Data Mining Group), an independent, vendor led consortium. Its main purpose is to represent data mining models, but also descriptions of data processing with various transformations (either before or after applying a data mining algorithm), in a XML-format. Thus, our initial thought was that PMML could be used as this unified standard for expressing the basic processing of data and the application of simple data mining algorithm, that could be needed for the project. And once expressing the desired model in PMML, then the same document would be imported by the different engines under evaluation.

However using PMML proved not to be possible for two main reasons. First, because the PMML support for these frameworks ranges from limited to non-existent. Only Spark is officially registered from DMG and merely as a PMML producer, meaning that Spark currently supports exporting models but not importing already created ones. On the other hand, Flink is yet neither a producer, nor a consumer officially. At the time of writing PMML support remains an unresolved issue. Second, as far as the actual processing of data is concerned, despite of PMML offering simple ways of data manipulation, the potentials are limited and as a consequence more complex transformations may not be supported, or might require the implementation of custom functions.

For all the aforementioned reasons it seems that it is not the ideal choice for this study, as handling the data would follow a MapReduce approach, more likely using custom functions and it can also be a barrier for including certain engines in the evaluation (e.g. Flink has no support whatsoever).

Chapter 6

Results

6.1 Memory usage

The purpose of the project was to provide an insight and evaluate the way each framework would exploit resources of a computing cluster, when executing the very same application. Starting from the case when the cluster consists of one worker node (along with the master node) and moving along by increasing the number of worker nodes, there are different results for the cluster as a whole, for the average worker node and for the master node.

To begin with, the memory usage of the cluster can be seen in Figure 6.1. Flink’s cluster memory usage is overall lower than in Spark and the difference keeps increasing as more nodes are added.

![Cluster’s memory usage (GB)](image)

*Figure 6.1. Memory usage of the cluster*

Flink’s master node memory usage has a slight increase as more nodes are added to the cluster as can be seen in Figure 6.2, however in all cases the absolute value is
smaller than in the case of Spark which reserves the same resources regardless of the number of nodes. What makes a difference however, is the memory usage on average in the worker nodes Figure 6.3. Flink’s nodes used a much smaller amount of memory as opposed to Spark’s nodes. While the absolute values kept decreasing in both cases, there was a slightly higher decrease rate for Flink.

![Master’s memory usage (GB)](image)

**Figure 6.2. Memory usage of the master node**

![Worker node’s average memory usage (GB)](image)

**Figure 6.3. Memory usage of the worker node on average**

This behavior is due to the fact that Flink integrates a sophisticated memory management mechanism which represents memory as Memory Segments where data are efficiently serialized in. The algorithms that run the tasks can therefore request for memory and release it. The same Memory Segments can be reused over and over again and this also has an impact on garbage collector, as there are no long-lived records but long-lived objects (Memory Segments) that are never being garbage collected.
On the other hand Spark has no explicit mechanism to handle memory. Since no
tuning has been done whatsoever, Spark does not exploit memory resource as effi-
ciently. RDDs are not serialized in the most effective way to save memory (Kryo\textsuperscript{1}
serialization could improve memory usage), while the garbage collector is not effi-
cient in identifying the lifecycle of objects and clearing old data as soon as possible.

Another interesting observation is that there was always a worker node that had
significantly higher memory usage than the others. The reason for that was that this
particular node had to hold the time window operators which cannot be done in paral-
lel. This is because each element of a stream must be processed by the same window
operator that decides which windows the element should be added to. This does not
happen in Spark because data are expressed as RDDs, which are collections that are
parallelized already and can be accessed by all the nodes of a cluster simultaneously.

6.2 CPU utilization

With respect to the CPU utilization, Flink exploited a higher percentage of the CPU
when compared to Spark, however as the number of nodes increased in the cluster, the
CPU utilization percentages of both dropped and also the difference between them
decreased as can be seen in Figure 6.4.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{cluster_cpu_utilization.png}
\caption{CPU usage of the cluster}
\end{figure}

Flink and Spark had different behaviors when it came to the master and worker level as seen in Figure 6.5 and Figure 6.6 respectively. Spark's master node had a
much higher CPU utilization as opposed to Flink master. The Spark master, also
known as the driver node, is responsible for translating every RDD into an execution
graph, scheduling the tasks and controlling their execution by the executors that are
found on the worker nodes. In other words the driver node has to perform scheduling
for every micro-batch that the data stream is split into.

\textsuperscript{1}https://github.com/EsotericSoftware/kryo
Flink programs on the other hand create a dataflow graph in the beginning which is followed accordingly from the JobManager (master node). Based on that graph the responsibility of the master node is to assign tasks to TaskManagers (worker nodes) and supervise the execution.

The average CPU usage is much higher on Flink worker nodes but as more nodes are added to the cluster it is substantially reduced, verging to the corresponding values of Spark which also has a very slight decrease with more nodes. The Spark workers however are still not using as much CPU as the Flink ones. The reason is that Flink essentially never stops processing records as it follows a native streaming approach where the operators have to be available constantly so that records can flow through them. On the contrary the transformations of Spark are executed following a certain interval (every 2 seconds in particular).
6.3 Network traffic

The total data network traffic rates for all the computers in the Cluster can be seen in Figures 6.7 and 6.8. In both cases while Flink starts having a lower rate with a small number of worker nodes, it eventually surpasses Spark’s rate as the number of worker nodes increases.

![Cluster's network traffic in (KB/s)](image)

*Figure 6.7. Incoming data rate of the Cluster’s computers*

![Cluster's network traffic out (KB/s)](image)

*Figure 6.8. Outgoing data rate of the Cluster’s computers*

Again when examining the master’s node behavior explicitly for both cases the difference is apparent as can be observed in Figures 6.9 and 6.10. Spark’s micro-batch approach results in a much more frequent communication with the master node, which is supposed to schedule and coordinate the execution of tasks in each micro-batch. In addition, for a task to be executed, the execution of all the previous ones must be finished so that the master node can assign tasks to executors accordingly.
Figure 6.9. Incoming data rate of the master node

Figure 6.10. Outgoing data rate of the master node

On the contrary, Flink has higher numbers in data going in and out of a worker node on average as shown in Figure 6.11 and 6.12. It is interesting that this happens after the second worker node is added to the cluster. This means that consecutive operators might reside on different machines, therefore the intermediate result and their partitions will need to be communicated from one worker node to another. Finally it is worth mentioning that the results vary between a few KB/s and a few MB/s, therefore the network capacity of our cluster is not affecting the performance of the algorithm according to the results we got in network measurements in chapter 5.1.2.
Figure 6.11. Incoming data rate of the master node

Figure 6.12. Outgoing data rate of the master node
Chapter 7

Conclusion

7.1 Discussion

One of the significant differences that was observed during the experiment was the memory usage of the two frameworks when they scaled out to a number of worker nodes. Flink and its memory management mechanism used less memory both in the level of the master node and the worker nodes. That indicates a much more efficient usage of memory from Flink's perspective as opposed to Spark (version 1.4.1), which has no explicit mechanisms for this issue apart from suggestions for tuning which has to be done in a user level. For the sake of this study there was no tuning whatsoever in either of the two frameworks. It is worth mentioning that Spark has introduced a memory management mechanism with project Tungsten\(^1\) which is integrated in version Spark 1.5.

Another noteworthy remark is the level of involvement of the master node for the two frameworks. In both cases the master/worker pattern indicates that scheduling and coordination is done by the master node, while the execution of the task and some basic communication with the master is what the worker nodes' job is. Nevertheless, due to its micro-batch nature, Spark needs its master node to communicate a lot more frequently with the clusters workers as it has to perform the schedule of each micro-batch and assign every task to the according executor. Flink on the other hand needs the master node to follow the JobGraph which is initially created from the Flink's program graph builder.

The work for Flink is more shifted to the worker nodes, as they are the ones responsible for carrying out all the tasks. Operations like time windows cannot be done in parallel in Flink and that combined with the high load that Flink workers can get, can become a problem when having a cluster with few nodes. However as seen in our experiment when adding more nodes in the cluster, this issue can readily be overcome.

\(^1\)https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
To conclude, the aforementioned observations indicate that Apache Flink would be better suited for the needs of this task. That is mostly because the master node is not as stressed as in Spark. Master node is a single point of failure and it can be caused by excessive utilization of CPU. Moreover Flink seems to be scaling out more gracefully than Spark as more worker nodes are added to the infrastructure. Last but not least, as far as other applications are concerned, when latency matters Flink is a better option since it provides results in real-time and there is no batch interval as opposed to Spark, therefore no additional mandatory waiting time.

7.2 Future work

The state-of-the-art analysis that was done in Chapter 2 focused mainly in a theoretical level for presenting the frameworks that were found. Later on the number of frameworks selected for a more thorough evaluation were two (Apache Flink and Apache Spark). Nevertheless any other framework of those in Chapter 2 could undergo the same extensive study and be eventually compared to Flink and Spark. Moreover it could be put in comparison either by using a similar benchmark, or by using more simplistic benchmarks.

Another way to extend the insight provided from this project would be a study and evaluation of other sections that Flink and Spark offer, such as applications including iterative algorithms, or graph processing. Both frameworks offer explicit libraries and extensions for these purposes and their differences have been indicated in this project.

Furthermore both Flink and Spark include several configurations and parameters which can offer tuning capabilities in terms of performance and memory and CPU usage. Therefore apart from altering the size of memory or the number of cores used by each node, there can be various ways to optimize them for coping with an application in a more efficient manner than with the default configurations. Moreover, apart from monitoring metrics such as memory or CPU usage, the creation of a benchmark that would aim to find the maximum throughput supported would be of great interest.

Finally, Spark has already announced Structured Streaming which will be introduced in Spark 2.0.\footnote{https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-faster-and-smarter.html} The Structured Streaming API will be largely built on Spark SQL engine. The vision behind that is that data will be perceived as continuous DataFrames (distributed collection of data organized into named columns) instead of RDDs. Thus the catalyst optimizer used by Spark SQL will be able to build a logical plan according to which queries will be executed on DataFrames incrementally and continuously.

Besides the utilization of the catalyst optimizer Structured Streaming brings more potential in terms of memory usage by exploiting project Tungsten \footnote{https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html} compact encoding. By using a feature called Encoder, in particular, Spark will be able to translate domain objects to its internal representation, providing that way greater space efficiency

52
than RDDs. Moreover Encoders allow for much faster serialization/deserialization as opposed to Java or Kryo serialization.

Structured Streaming brings therefore a lot of potential and attempting a similar experimentation as in our study would be quite insightful both for illustrating the improvements of Structured Streaming as opposed to Spark Streaming and for comparing a new approach of data stream processing against the approach presented by Flink.

7.3 Limitations

There were no major limitations during this project. A few minor issues were that the initial thought of describing the code of the experiment using PMML so that we could achieve interoperability to other frameworks did not prove to be feasible. Also for ingesting data into our applications, we selected socket sources but during the study it proved that could not be parallelized. Therefore, to make sure that this would not become a bottleneck, new custom socket sources were created and used.
Glossary

**AMPLab** Algorithms-Machines-People Laboratory in University of California, Berkeley.

**API** Applications Programming Interface.

**EPL** Event Processing Language.

**ETL** Extract-Transform-Load.

**HDFS** Hadoop Distributed File System.

**JDBC** Java Database Connectivity.

**JMS** Java Message Services.

**JVM** Machine Learning.

**RDD** Resilient Distributed Datasets.

**YARN** Yet Another Resource Negotiator.
References

[1] [n.d.].

URL: http://research.google.com/pubs/pub43864.html

URL: http://dx.doi.org/10.1007/s00778-014-0357-y

URL: http://doi.acm.org/10.1145/2523616.2523629

URL: http://doi.acm.org/10.1145/2723372.2742797

URL: http://arxiv.org/abs/1506.08603

URL: http://doi.acm.org/10.1145/214451.214456
URL: https://databricks.com/blog/2015/07/30/diving-into-spark-streamings-execution-model.html


URL: http://doi.acm.org/10.1145/1327452.1327492

URL: http://dx.doi.org/10.14778/2350229.2350245


URL: http://www.apress.com/9781484209653

URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7384369


URL: http://amazon.com/o/ASIN/1449329705/

URL: http://doi.acm.org/10.1145/1107499.1107504

URL: http://www.infoq.com/articles/stream-processing-hadoop

URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html

URL: http://dl.acm.org/citation.cfm?id=2228298.2228301

URL: http://doi.acm.org/10.1145/2517349.2522737