HAW HAMBURG

Bachelorarbeit

Alexander Könemann

Experimental comparison between Apache Spark and Flink in heterogeneous hardware environments

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Alexander Könemann

Experimental comparison between Apache Spark and Flink in heterogeneous hardware environments

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 14.07.2022

Alexander Könemann

Thema der Arbeit

Experimenteller Vergleich zwischen Apache Spark und Flink in heterogenen Hardwareumgebungen

Stichworte

Apache Spark, Apache Flink, Stapelverarbeitung, kontinuierliche Verarbeitung, Raspberry Pi, Verteilte Datenverarbeitung, Leistungsvergleich, Big Data

Kurzzusammenfassung

Apache Spark und Flink werden kommerziell vorwiegend in Rechenzentren mit hochperformanten Computern eingesetzt. Ein gänzlich anderes Szenario stellt der Einsatz von heterogener Hardware dar, welches in dieser Studie betrachtet wird. In verschiedenen Versuchsaufbauten wird die Datenverarbeitung getestet und die Leistungsfähigkeit beider Systeme analysiert. Dafür wurden fünf Hypothesen aufgestellt und betrachtet. Es konnte gezeigt werden, dass das Hinzufügen von zu schwacher Hardware einen negativen Einfluss auf die Leistung eines Clusters hat. Weiterhin hat die Leistungsfähigkeit der Master Node einen signifikanten Einfluss auf die Gesamtleistung. Beim Systemvergleich schnitt Spark besser in der Stapelverarbeitung ab, wohingegen sich Flink bei der kontinuierlichen Verarbeitung überlegen zeigte.

Title of Thesis

Experimental comparison between Apache Spark and Flink in heterogeneous hardware environments

Keywords

Apache Spark, Apache Flink, batch processing, stream processing, Raspberry Pi, Cluster Computing, Benchmarking, Big Data

Abstract

Apache Spark and Flink are primarily deployed in commercial data centers on high-performance nodes. A fundamentally different approach is the utilization of heterogeneous hardware, which is considered in this study. In various experimental setups, data processing is being trialed and the performance of both systems is being analyzed. For this purpose, five hypotheses were formulated and investigated. It was shown, that insufficient hardware has a negative influence on a cluster. Additionally, the performance of the master node has a significant influence on the overall performance. Upon comparing both frameworks, Spark showed better performance in batch processing, whereas Flink was found to be superior in stream processing.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The last decades showed an immense growth of stored and processed data worldwide. In 2020 it has reached 64 Zetabytes and is projected to increase to 180 Zetabytes in 2025 [25]. The field of Big Data, which emerged from this trend, deals with technologies for processing huge amounts of data. The landscape of technology for processing and storing big data is changing very rapidly. In general, a trend toward distributed storage and processing of data can be observed.
The first big player for distributed processing, Hadoop, with its famous map-reduce framework is already being replaced by newer, faster and more reliable technologies. Apache Spark has gained a lot of popularity recently and Flink with its new streaming approach is a promising candidate as well. These technologies are generally deployed in data centers, which consist of numerous, powerful nodes. The construction and operation of these data centers incurs high costs and consumes valuable resources.
A completely different approach is to reuse second-hand hardware instead of deploying homogeneous high-performance nodes. However, this gives rise to new problems in the configuration and maintenance of a cluster, as well as the question to which extent weaker nodes can affect the overall performance. In this respect, the suitability of Apache Spark and Flink on heterogeneous hardware will be investigated.

1.2 Objectives and Scope

Evaluating the application of Apache Spark and Flink on budget hardware is the primary focus of this elaboration. This is a rather unusual approach, since these technologies are usually applied on high-performance and homogeneous hardware. However, it could lead to new areas of application, from private computing clusters to the recycling of old computers in large data centers. The general feasibility on different nodes in a small cluster

(6 nodes) shall be tested in this respect.
At first, a brief introduction of Apache Spark and Flink will be given, followed by a theoretical comparison of both frameworks. This foundation will be used to develop a cluster environment as well as for the development of test applications for a performance comparison. In order to systematically compare both frameworks, several hypotheses will be presented concerning the influence of hardware capabilities on the performance on the cluster and conceptual differences between both frameworks. To challenge the hypotheses several experiments will be conducted. The design and setup of these benchmark tests will be laid out in the course of this elaboration. The hardware used differs mainly in terms of different processor technology and performance as well as the size of available working memory.
The main processing principles of both frameworks will be the subject of comparison, namely batch and stream processing. In order to compare these basic processing principles, separate performance metrics will be defined. These include the runtime for processing a given dataset for batch processing and the maximum throughput for stream processing. Finally, a general suitability of the frameworks in the specific context will be derived and an outlook on further research questions will be given.

1.3 Related Work

In the scientific community there have already been elaborations that have dealt with a comparison between Apache Spark and Flink. These works will be briefly introduced and a contextualization of this study will be given to outline the particular features of this study.

Extensive comparative experimental studies between both frameworks have been done by Kaepke [18] and Masurat [23]. The former focuses his research on graph analyses in a big data context and the latter on real-time detection of hate speech. Both are conducting their experiments on a homogeneous 4-node cluster.
Furthermore, scientific papers can be found which are focusing on benchmarking and scalability in a multi-node cluster environment. Garcia-Gil et al. [15] published a study on comparing the scalability for batch big data processing on Spark and Flink. This concise survey performs benchmark tests with a focus on the machine learning libraries of each framework yielding better performance for Spark.
Marcu et al. [22] conducted a detailed analysis of performance differences for big data

analytics. They tested a variety of application scenarios with different configuration parameters in a multi-node cluster. As a result, it was shown that no framework is generally superior, but it depends on the respective data type, use case and configuration parameters.

All studies share the trait, that they examine both frameworks while focusing on specific use cases and scenarios. However, the number of deployed worker nodes varies greatly from a few machines to large clusters. Only homogeneous hardware is used in these scenarios. This study will focus on exploring the usage of heterogeneous hardware. In this course, it shall be investigated whether Apache Spark or Flink is more suitable for this purpose. To clarify this matter, various use cases and metrics are used as reference.

2 Conceptional foundations

In order to conduct an empirical comparison between Apache Spark and Flink, each representing a cluster data-processing framework, both technologies will be introduced and compared theoretically at first. Subsequently, this basis will be used to develop and apply a methodology for comparison.

2.1 Apache Spark

Apache Spark is an open source cluster computing platform designed for general-purpose big data batch and stream processing tasks with high performance [4]. It first started as a research project at the University of California (Berkeley), back in 2009 and open sourced in the following year [11]. The source code of the repository can be accessed on GitHub [13]. Apache Spark is well covered in academic literature. A broad overview of the core aspects as well as additional practical applications can be found in [1], [4], [14] and [19].
Essentially, Apache Spark implements the MapReduce processing model, focusing on robust in-memory processing to provide increased speed over its predecessor technology, Hadoop [19]. Furthermore, it provides the developer with a broad range of APIs in Scala, Python, Java, R and SQL as well as possibilities for data source integrations such as Amazon S3. In order to make an empirical comparison between Apache Spark and Flink, the main concepts, which are illustrated in figure 2.1 will be introduced.

2.1.1 Architecture

Spark has been designed with a master-slave architecture with a cluster manager [4], as shown in figure 2.2. Spark applications are executed as independent sets of processes on a cluster, coordinated by a driver program using a SparkContext object. The SparkContext is an entrypoint for Spark applications and is programmatically available as 'sc'.

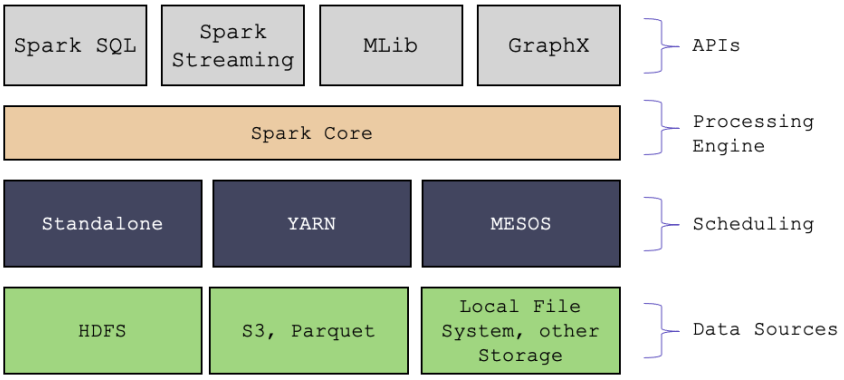

Figure 2.1: Spark components with APIs, Processing Engine, Scheduling and Data Sources (own illustration based on [19] and [4])

In order to run an application on a cluster, the SparkContext connects to a cluster manager, which allocates resources across worker nodes [10]. Once the connection has been established, executors on worker nodes are being acquired. Executors are processes that run computations and store data for applications. In the next step, the applications source code is sent to the executor, e.g. by passing a JAR file. Finally, the SparkContext sends tasks to the executors to run. The core aspects of the Spark Driver, the executors and the Cluster Manager will be covered in this section.

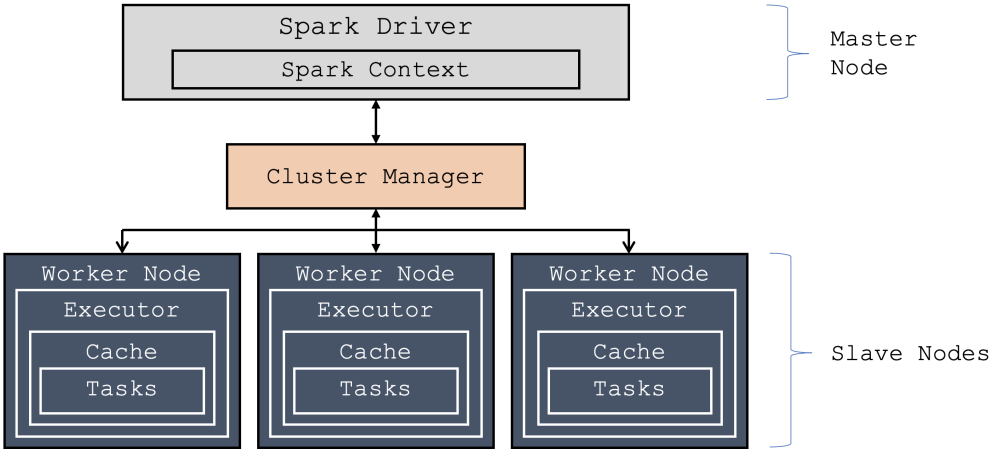

Figure 2.2: Spark architecture with master and slave nodes forming a cluster (own illustration based on [19] and [4])

Key characteristics of the Spark Driver [4], [19]:

- The SparkContext is created within the driver program. Upon starting a Spark interactive shell, a driver program and SparkContext will be created and kept alive until the shell is being terminated.
- It runs the application's main function.
- The Spark driver is responsible for converting user applications into execution units called tasks. Tasks themselves represent the smallest unit of work in Spark. A program can be composed by thousands of individual tasks. A task may have side effects such as caching data on an executor or a worker node respectively.
- It is responsible for scheduling and coordinating jobs as well as allocating resources for execution.
- It has a complete overview of all available executors and their resources as well as the location of currently cached data on the executors. Cached data can be used for efficient scheduling of future tasks.
- It exposes information about the cluster and applications through a web interface, which is by default available at [http://localhost:4040].

Key characteristics of the executor(s) [4], [19]:

- Upon starting an executor, it registers itself with the driver.
- It is responsible for performing the actual workload of a program by executing tasks and performing data processing. It returns the results to the driver.
- Providing in-memory storage for RDDs (see 2.1.3 for further information on data structures) that are cached by user programs through a Block Manager.
- If an executor fails during a job the application itself can still be running and the tasks might be assigned to different workers for execution.

Key characteristics of the Cluster Manager [4], [19]:

- Spark uses the Cluster Manager to launch the executor and driver processes. It is responsible for acquiring resources (e.g. CPU and memory) and allocating them to Spark applications.

- The Cluster Manager is a pluggable component in Spark. Spark runs on top of different external Cluster Managers. There are four available types: Standalone Deploy Mode, Hadoop YARN[1], Kubernetes and Mesos[2] (deprecated). Different Cluster Managers are useful if the resources of a cluster are shared by different Big Data platforms such as Hadoop MapReduce or Apache Flink. Detailed instructions and examples on Cluster Managers can be found in the official Spark documentation, see [12].
- In order to execute a user program on the cluster a script is provided by Spark which is called 'spark-submit'. Through the setting of options it can connect to the respecting Cluster Manager (e.g. the process can run on a YARN worker node). It is also possible to pass additional arguments to specify resources to be used on the cluster.

The minimum setup of a cluster requires a Spark Driver with a Cluster Manager which is connected to at least one executor node in order to execute Spark applications. Scaling a cluster, requires adding further worker nodes or replacing them with more capable ones.

2.1.2 Libraries and APIs

To outline the scope of the Spark platform, a brief overview of the main components will be given (see [4], [12] and [19] for reference). The Spark stack which has been introduced in figure 2.1 contains multiple closely integrated components. The central component is the Spark Core which is responsible for scheduling, distributing and monitoring of applications. On top of the Core several higher-level components are available, which have specialized purposes. Since these components are integrated in the project it is rather simple for a developer to include any of these libraries or migrate to newer releases.

Spark SQL

Spark SQL is a module for structured data processing. It can be utilized to query in SQL or HiveQL. Potential data sources include Hive tables, JSON and Parquet. The interface also offers the possibility to combine SQL Queries with the common RDD API and thus

[1] Yet another resource negotiator
[2] A general-purpose cluster manager

provides the flexibility to combine the features of each library. Spark SQL represents a newer interface than the use of RDD. The advantages of the API include more extensive information about the structure of the data and accompanying internal optimisation of the actual execution of the user program. If the API is used, the data representation will be of type Dataset or Dataframe. Possible data types and their characteristics are discussed in chapter 2.1.3.

MLib

Spark offers an extensive machine learning library called MLib. The main objective of this module is to provide scalable, robust and distributed machine learning algorithms. Common ML algorithms such as classification, regression, clustering and collaborative filtering are provided. It also provides supporting functionality, such as tools for constructing, evaluating and tuning of ML pipelines. Additionally tools for saving and loading models, data import and model evaluation are provided. The DataFrame API is the central data structure when applying this library.

GraphX

GraphX is an extension for creating and manipulating graph data structures by performing graph-parallel computations. The library extends the Spark Core by the option to create directed multigraph abstractions with arbitrary properties for each vertex and edge. The library also provides operators for graph manipulations such as to create subgraphs or join vertices as well as a collection of graph algorithms for analytical purposes. The underlying data structure is extending RDD by vertex (VD) and edge (ED) types. Graphs can be partitioned across the executors for distributed computing.

2.1.3 Data structures

Spark currently provides three distinct types of data structures. The key characteristics as well as major differences will be outlined in this section and provide the basis for later use in the practical case study.

Resilient Distributed Dataset (RDD)

The fundamental data structure of Spark or the Spark Core API respectively is known as a resilient distributed data set (RDD) [4]. Until Spark version 1.3 it has been the only available data structure. A RDD is an immutable, fault-tolerant collection of objects that can be operated on in parallel.
RDD share the following characteristics (see [19], [4], [12] and [5]):

- In-memory computation: Per default RDD store intermediate results in distributed memory. This is an advantage of Spark over its predecessor Hadoop, which stores results on disk, which is considerably slower.
- Partitioning: RDDs are divided into partitions and distributed across the cluster. This means, that only a partition of the data set will be stored on a given node. The number of partitions can be specified. Spark will execute one task for each partition on the cluster. The Apache Spark Documentation Guide recommends 2-4 partitions for each CPU in the cluster. It is also possible to persist an RDD (cache) on a worker node or even replicate the same RDD on several machines to avoid a slowdown if the RDD has to be recalculated on failure.
- Fault-tolerance: If a RDD is lost, e.g. because a worker node crashes, it can be rebuild automatically from the source of failure using lineage. Each RDD stores the information how it has been created from parent data sets. Per default data is stored in-memory which can be supplemented by storage on HDD (serialized or unserialized). The advantage of using serialized data objects is an increase in space efficiency at the cost of higher workload for reading and saving file objects. Additionally this function is only available in Java and Scala.
- Immutability: A RDD is immutable and therefore cannot be modified in-place. It can only be modified by applying RDD operations, namely transformations and actions.
- Transformations: A transformation creates a new RDD from an existing input RDD. Filtering or mapping on a data set are common examples for a transformation. Spark does lazy evaluation on transformations and therefore only computes results when they are needed (on an occasion when an action is carried out).
- Actions: When an action is called the lazy evaluation chain is being executed. An action takes a RDD as input and transforms it into non-RDD values. An example

of an action in Spark is to count the number of occurrences of a given key in a collection. Only by using actions, data can be send from an executor to the Driver e.g. by using the collect method which returns the RDD to the Driver program on the master node. The limitation of the collect action is, that the machines must have sufficient resources to process the resulting data volume.

- Statically typed: A RDD has an explicitly declared type (e.g. RDD[String]) and thus its type is determined at compile time.
- Accessible interface: RDDs can be accessed via various programming languages as Spark provides Scala, Java, Python and R APIs.

In conclusion, RDD provide the developer with a wide range of options to perform robust and in-memory computations on large clusters. However there are some limitations to this data structure. The structure of an RDD can be considered as a black box, which does not provide information about the data it stores. Therefore internal optimisations are limited due to missing insights about the enclosed data as well as higher-level operations on the data. Another disadvantage is, that it is incompatible with the SparkSQL API. It is considered to be the preferred option for unstructured data, to be used for low-level transformations and actions [26].

DataFrame

A DataFrame is representing a distributed dataset based on RDD with some additional features and is available since Spark version 1.3. It is a collection of data organised into named columns and conceptually similar to a table in a relational database [26] or a data frame in R or Python [12]. Compared to RDDs, DataFrames provide Spark with more information about both the data and the calculations. These are used for internal data processing optimisations [12]. Data Frames can be constructed from existing RDDs, Files or external databases. It has the same accessibility as conventional RDDs (see 2.1.3) and is represented by a Dataset of Rows (Dataset[Row]).
One Advantage of this data structure is a more extensive user interface for interacting with data. It offers high-level operations including specific operations on a given column instead of the whole data structure. Depending on the use case it can also provide better performance by reducing the cost of loading the data and optimisation of execution plans. Compared to RDDs, DataFrames do not provide compile time type safety.

DataSet

A Dataset represents the third and newest distributed collection in Spark and is available since version 1.6. It aims to merge the advantages of RDDs (compile time safety) and DataFrames (higher-level operations on data and internal execution optimisations) [12]. Compared to RDD and DataFrames there is currently no support for Python and R. DataSets are also considered to be slower on execution [26].

In conclusion, after introducing the various data structures and their differences, it can be said that no general recommendation can be given for any of the types. The choice of an appropriate data structure is highly dependent on the application. When applied in a classic Word Count use case, conventional RDDs are useful, but when complex data transformations and aggregations of e.g. hive tables are to be performed, DataSets are the preferable option.

2.1.4 Stream and batch processing

Spark supports two distinct categories of processing principles: Stream and batch processing, the latter represents the default behaviour.
In the case of batch processing, data is collected over a period of time prior to evaluation and then processed collectively or in smaller units [4]. The core aspect here lies in the fact that the data must be entirely available prior to processing. Therefore the disadvantage of this model is that only retrospective analyses can be conducted.
The general workflow of a simple program using batch processing can be reduced to the following steps [19]:

1. The execution of a Spark program starts when a user submits an application to the cluster by using spark-submit.

2. The spark-submit procedure then launches the driver program and invokes the main() method which has to be specified in the deployed application as an entry point.

3. The driver program contacts the cluster manager to acquire resources to launch executors on worker nodes.

4. The Cluster Manager then launches executors (number can be specified) on behalf of the driver program.

5. The driver process now executes the user application. Based on RDD actions and transformations to be executed by the program, the driver sends chunks of the workload to executors in the unit of tasks.

6. Tasks are being run on executor processes on worker nodes to perform computations and save or return results.

7. Whenever the driver exits or stops the SparkContext, it will terminate the executor processes. The resources of the respecting worker nodes will be released by the cluster manager.

However, in the batch processing scheme that has just been described, only data that is already available can be processed. There are use cases where insights about current events or real-time processing are required, where data is being processed when it is generated. This principle is called stream processing and will be focused on in this section due to the fact, that Spark Streaming is internally treated as a special case of batch processing [4]. A typical use case of streaming is to analyse Tweets on Twitter or tracking of user interactions on websites for optimising product recommendations [4], [19].
Spark offers two APIs for Streaming which are called DStream (Discretized Streams) or Structured Streaming. Both are extending the Spark Core and offer a scalable, fault-tolerant and event-oriented streaming processing system [12].
The internal processing structure is an interesting extension of the basic batch processing principle. For processing real-time data, the batch processing model is used, but the incoming data is divided into microbatches [4]. The short time interval between two batches thus approximates the behaviour of a stream.
DStreams consist of a series of time-ordered RDDs, each of which contains the data of a specific time interval. The lower bound for a viable time interval is currently at 0.5 seconds. The exemplary illustration of a DStream is shown in figure 2.3. The processing of four microbatches with a time interval of 1 second is being illustrated.

Spark Streaming offers various options for connecting input data sources. The Spark Streaming API can be supplied with data e.g. via TCP sockets, Kafka, HDFS and Twitter. To apply a DStream in a Spark program an additional StreamingContext has to be used within the Driver program (in addition to the SparkContext), which is responsible

Figure 2.3: Spark DStream as continuous, time-ordered series of RDDs, each containing one second of data (own illustration based on [19] and [4])

for periodically executing jobs to process new microbatches in a specified time interval [4]. For each input source, a receiver will be launched on an executor process. A receiver collects input data and stores it as RDDs. By default these RDDs are replicated to other executor nodes to provide fault tolerance. The data storage is similar to cached RDDs and will be stored in the memory of the executors.
The second spark model for stream processing is called Structured Streaming and will be briefly introduced. More information on streaming can be found in [12], [19], [4]. Spark Structured Streaming is as an API which is build op top of Spark SQL [4]. The processing model is similar to DStreams as it is also using a microbatch processing engine. In contrast to RDDs the Dataset/DataFrame API is used to transform the data stream into an unbounded table where each arriving data item is being appended as a row to the table (see figure 2.4 which is illustrating the concept).

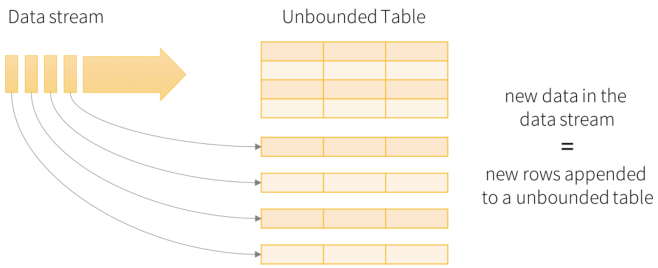

Figure 2.4: Spark Structured Streaming programming model as an unbounded table (see [12])

The advantage of this model is, that the user can express queries as if they would be performed on static data, which will be incrementally processed by the Spark SQL engine. Another advantage is, that the time interval of a batch is considerably lower and currently at 100 milliseconds [12].

2.2 Apache Flink

Apache Flink is an open-source cluster computing platform designed for distributed stream and batch data processing [6]. The core objective of Flink is to process data streams in real time (low data latency) while being able to process high volume of data in a fault-tolerant manner [17]. It started as a research project at the Technical University in Berlin back in 2009 and open sourced in 2014 [17], [2]. The source code of the Flink project's repository can be accessed on GitHub [9].
Apache Flink is not widely covered in academic literature. A brief introduction can be found in [17] as well as in the official documentation [6]. In addition, there are a variety of papers published, that are rather focused on particular subject areas, such as [15], [3]. Flink provides a variety of developer APIs for distributed data processing in Scala, Java, Python and SQL as well as typical options for data source integrations such as Kafka or Hive [6]. Similar to Apache Spark, which has been introduced in section 2.1, the main concepts of Flink will be outlined as a foundation for a subsequent comparison of both frameworks. An overview of the main components of Flink is illustrated in figure 2.5.

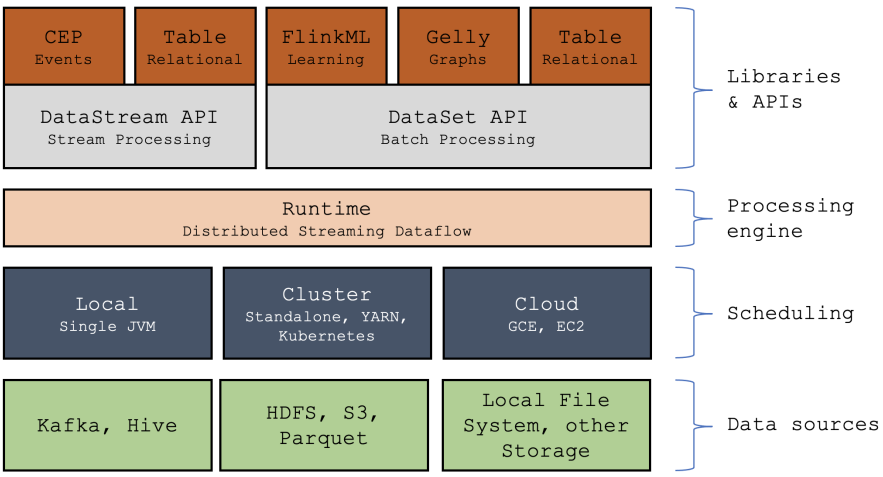

Figure 2.5: Flink components with APIs, Processing Engine, Scheduling and Data Sources (own illustration based on [6] and [3])

2.2.1 Architecture

Flink has been designed in a master-slave architecture with a Job Manager and one or more Task Managers [6], as shown in figure 2.6. In order to execute a Flink application

four different components need to work together. These components, collectively forming a cluster, are called Job Manager, Resource Manager, Task Manager and Dispatcher [17]. As Flink is implemented in Java and Scala, the runtime environment is bound to Java Virtual Machines. The execution of a program is coordinated by the Job Manager and programmatically controlled by the ExecutionEnvironment which can either be a local or cluster environment [6].
To run an application, a client needs to submit an application to the Job Manager, e.g. by passing a JAR File. The Job Manager controls the execution and creates an Execution Graph, which contains tasks that can be processed in parallel. Prior to an execution the Job Manager acquires resources, namely Task slots on Task Managers. Once sufficient resources have been allocated, tasks are being distributed and executed on dedicated worker nodes. The core aspects of the Job and Task Manager will be covered in this section.

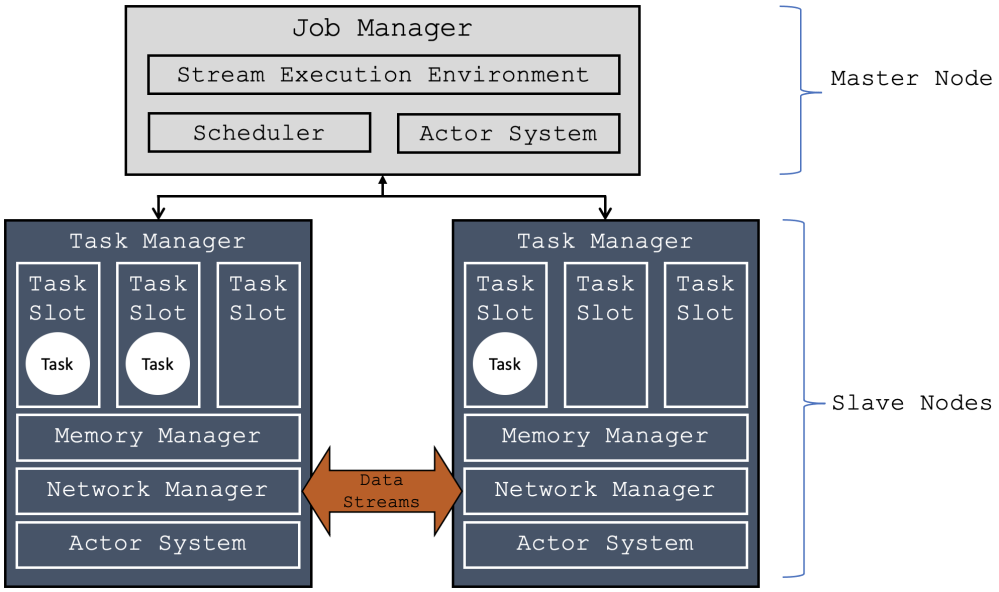

Figure 2.6: Flink architecture with master and slave nodes forming a cluster (own illustration based on [6])

Key characteristics of the Job Manager [17], [6]:

- The Job Manager is the master process that is responsible for controlling the execution of an application. Depending on the cluster mode, each application might be controlled by a different Job Manager (Flink Job Cluster mode). The cluster mode determines its life cycle. Alternatively, a long running Session Cluster can be

spawned, that can accept multiple job submissions. A third option represents the Application Cluster mode where the lifetime is bound to the lifetime of the flink application.

- When it receives a JAR file for execution it runs the application's main function. An application is structured by a Job Graph, which determines the logical flow of the program. The Job Manager converts the Job Graph into an Execution Graph, that is comprised of parallelized tasks. These tasks will be distributed to assigned Task Managers.
- It is responsible for coordinating tasks, which is the unit of work in Flink, as well as allocating resources on Task Managers for execution. The respecting component is called Resource Manager and it schedules Task Slots for processing an application. There are a variety of different Resource Managers available, e.g. YARN, Mesos or Standalone deployment. If more Task Slots are requested than idle slots available, the Resource Manager can be configured to request a resource provider to spawn new Task Manager containers.
- It is responsible for central coordination actions, such as saving checkpoints. Checkpoints are used for recovery and storing the state of data.
- It exposes information about the cluster resources as well as the option to submit applications through a web interface, which is by default available at [http://localhost:8081]. It is also possible to use this interface for submitting JAR files for execution. The interface for submitting jobs is provided by the Dispatcher.

Key characteristics of the Task Manager(s) [17], [6]:

- A Task Manager is the worker process which is responsible to perform dedicated chunks of the actual workload, namely tasks.
- There can be a multitude of Task Managers each connected to a Job Manager, providing their resources in the form of Task Slots.
- Each Task Manager provides a specified number of Task Slots, which limit the number of tasks it can execute in parallel. Depending on the configuration, each slot may be assigned to different applications (job parallelism).

- While conducting an arbitrary task, Task Managers that are running the same application might exchange data with each other. To reduce network traffic, it might be useful to schedule closely related tasks on the same Task Manager to isolate resources. However, each task is executed as a lightweight Thread in the same JVM process on a Task Manager, so the consequences of a failing task is higher and thus represents a trade off for the developer.
- To provide a robust cluster setup, it is crucial to provide a sufficient number of processing slots. If a Task Manager fails, the Job Manager will try to acquire new resources trough the Resource Manager. If none are available, the application cannot be restarted. The restart strategy can be customized according to the needs of the given application.
- The number of Task Slots per Task Manager is limited by the resources of the underlying machine. The available memory will be distributed evenly between them. As of writing, CPU isolation between slots is not available.

The interaction of the components, which are triggered when an application is submitted to the cluster is illustrated in figure 2.7. The process starts by submitting an application to the Dispatcher e.g. by using the command line interface provided by Flink (by executing [.bin/flink run $PathToJar]). The Dispatcher in turn starts a Job Manager, which controls the execution of the application. At this point the core of the execution is triggered by acquiring Task Slots from the Resource Manager and submitting tasks for execution. During execution data might be exchanged between tasks.
Alternatively an application can be bundled in a container, such as a Docker image [17]. When the container is started the corresponding actions will be triggered automatically and can be externally controlled, e.g. by Kubernetes.

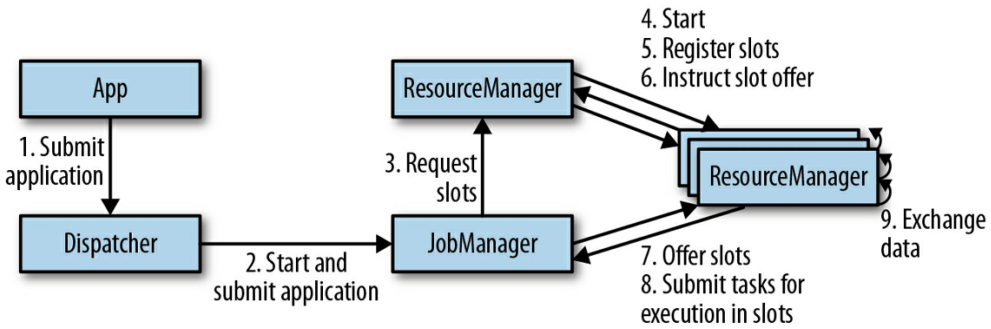

Figure 2.7: Interaction of Flink components upon submission of an application [17]

2.2.2 Libraries and APIs

To provide an idea of the scope of Flink's available components, a brief overview of the main components will be given (see [17], [7] and [6] for reference). The Flink stack and its integrated components have been introduced in figure 2.5. The central component is the Flink runtime which is a distributed data streaming engine. It is responsible for scheduling and running jobs either in stream or batch processing mode. On top of the Runtime reside the two main processing APIs, called DataStream and DataSet Api. The former is responsible for processing potentially infinite streams of data and the latter for processing bounded batch data sets. Both will be covered in depth in chapter 2.2.3. At the top of the stack there are several higher-level libraries available, each serving a specialized purpose.

FlinkCEP

FlinkCEP is a module for Complex Event Processing (CEP) and can be used to detect patterns in an endless stream of events. It allows to specify custom patterns, which abstractly describe the data to be detected and also actions that are triggered upon matching event sequences. A simple use case would be to scan an incoming event log stream for a specific event name and trigger an alert upon a potential detection.

Table

The Table Api offers a relational SQL related language for stream and batch processing. Due to its unified structure, no modifications are required when changing the processing mode (e.g. from stream to batch processing). Unlike common SQL queries which are represented as Strings, the Table Api offers language-embedded query definitions in Java, Scala or Python. Thereby compile time type safety, syntax validation and autocompletion can be supported by an IDE.

FlinkML

FlinkML is a library, that provides tools for building machine learning pipelines. It can be used to facilitate provisioned ML algorithms for building training and inference jobs. The provided algorithms include logistic regression, k-means, k-nearest neighbors, naive

bayes and one-hot encoder (see [24]). It also provides functionality for online learning of ML models as well as Python support in a recent release. Apache Flink's ML repository, which is mainly written in Java, has recently moved to a separate repository on Github (see [8]).

Gelly

Gelly is an extension for creating, processing and manipulating graph data structures. It can be used for directed or undirected graphs. The API provides a library of graph algorithms, e.g. Community Detection for detecting well connected groups of nodes. A Graph is represented by two DataSets, one containing vertices or nodes (DataSet[Vertex]) and the other edges (DataSet[Edge]) of the respecting graph. Each node has a unique ID.

Each of the provided Flink libraries serve a specialized purpose pursuing the goal of offering not solely a cluster compute framework that implements the map-reduce model, but also supplying integrated libraries for common use cases.

2.2.3 Data structures

As of writing, Flink provides two different types of data structures, namely DataStream and DataSet. The former is intended for the processing of bounded and unbounded streams while the latter can be applied for bounded data sets [7]. In order to provide a basis for an application in a subsequent comparative study (see section 4), the key characteristics and major differences will be introduced in this section.

DataStream

A DataStream is the main data structure in the Flink ecosystem. It represents an immutable collection of data in a Flink program. The corresponding data can be finite or unbounded [7]. The data structure is similar to a Java Collection, but it can just be inspected or manipulated using DataStream API operations, which are referred as transformations. Flink does lazy evaluation on transformations, that can be explicitly triggered by calling the execute method. In order to create a DataStream object, a source has to be specified, e.g. by passing a reference to a text file. After applying a

transformation on a DataStream, e.g. by mapping a collection, a new DataStream will be derived.
Partitioning can be implicitly achieved by applying the KeyBy method, which partitions a stream into disjoint partitions (internally implemented with hash partitioning) [7]. Low-level custom partitioning control is also available, where one can define the number of partitions after a transformation.
Recovery from failure is achieved through checkpointing [7]. Upon failure Flink will restart all running tasks from a checkpoint when executing in stream processing mode. In batch mode backtracking to previous stages will be used, if intermediate results are available. With this technique only tasks that actually fail will be restarted. This behaviour might be more efficient, if frequent failures are to be expected and batch mode can be applied.
DataStreams have an explicitly declared type, e.g. DataStream[String].

DataSet

A DataSet is only applicable on bounded data sets, where the data is already available at execution time. DataSets share the same features as DataStreams, namely lazy evaluation, partitioning, recovery upon failure and statical types (see 2.2.3). Starting with Flink 1.12 the DataSet API has been soft deprecated [7]. As of writing, the latest stable release version is 1.14. It is recommended to either use the DataStream API or use higher level libraries such as the Table API or SQL instead (see chapter 2.2.2 for reference). The decision for a future removal was made to offer as few APIs as possible in order to reduce the complexity of the framework [20]. The DataSet API is the older data type and was originally designed for an early stage of Flink, when it was meant to mainly become a batch processing framework on bounded data.

In conclusion, after introducing the data structures of Flink, DataStream types can generally be recommended since it's the only available data structure apart from the Table API and SQL in Flink. The latter are applicable for higher level transformations. Flink offers a comprehensive, yet flexible amount of APIs which serve a broad range of use cases.

2.2.4 Stream and batch processing

Flink offers two distinct processing principles: Stream and batch processing. The former represents the default behaviour. The general flow of a simple program, using any processing principle, consists of at least these basic parts [7]:

1. Acquire an execution environment, which is required for running a program on a cluster (or local machine).
2. Load or create initial data to specify a data source.
3. Apply transformations on the data according to the use case.
4. Specify, what shall be done with the results, e.g. write the data to an outside system by creating a sink.
5. Trigger the program execution to actually perform the lazy evaluations (transformations and loading of data).

All applications are composed of streaming dataflows [7], which are illustrated in figure 2.8. A dataflow is a directed graph which starts with one or many source nodes, may be transformed by Operators and end in one or more sink nodes. In practice this logical graph will be executed in parallel and chunks of the data distributed and potentially copied across multiple nodes in the cluster.

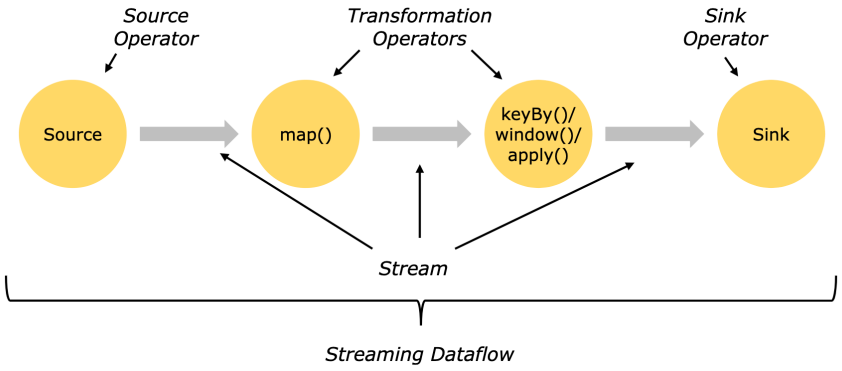

Figure 2.8: Flink's streaming dataflow with source, transformation and sink [7]

Stream processing

Flink's core feature is processing scalable and in-memory computations over unbounded and bounded data streams [7]. In general most data is produced as a stream of events, e.g. credit card transactions or user interactions on websites. In the case of stream processing this data can be processed as an unbounded stream. An unbounded stream has a start but no defined end and provides new data as it is generated. Due to the fact, that data is produced continuously, current events must be processed promptly to avoid a buffer overflow. In many cases, a specific order of processing is required, e.g. the order in which the events occurred [7].
Flink offers various features for timely stream processing, when aggregations are based on certain time periods (namely windows) or processing depends on the time when an event occurred [7]. The different notions of time are illustrated in figure 2.9 and include event and processing time. Processing time is the simplest notion of time [7] and refers to the system clock of the machine that is executing the corresponding operation. Event time on the other hand is the time that each individual event occurred and is created on the device which is producing the data. Typically, the event timestamp is retrieved from each event record. Due to its nature, event time causes latency in order to be able to wait for out-of-order events.

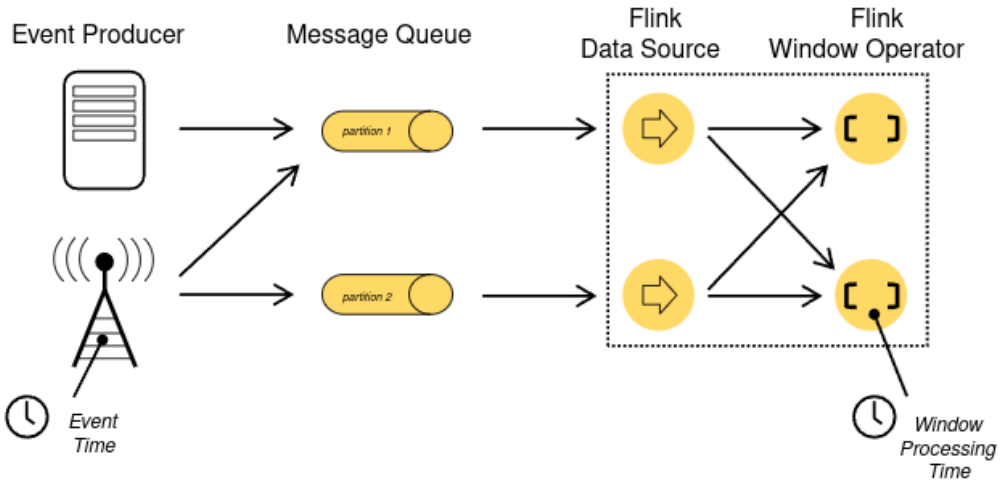

Figure 2.9: Flink's different notions of time for timely stream processing [7]

A time window can also be specified for aggregations of events, e.g. to count the occurrence of events in the last hour. It is also possible to specify how to treat events, that

arrive late to processing which might occur due to network latencies.
Additionally, operations can be stateful, e.g. by accumulating the count of a certain key and thus creating a dependency between events. Since applications are executed in parallel each instance has to be able to work independently. In order to process stateful operations each worker node in the cluster is responsible for handling events for a specific group of keys and keep the state for those keys locally [7]. State is kept locally, to achieve high throughput and low-latency. By default, state is organized on the JVM heap, but can instead be stored on-disk data structures as well [7].

Batch processing

In the case of batch processing bounded DataStreams are being evaluated, which have a defined start and end. Flink simply treats a batch data set as a finite stream of data, which allows alternating between processing modes conveniently without adapting the underlying data structure. Another way of conducting batch processing is by facilitating DataSets. But as mentioned in section 2.2.3 DataSets are deprecated and therefore DataStreams should be used in stream and batch mode instead.

2.3 Spark versus Flink

After introducing the core concepts of Apache Spark and Flink in the previous sections, their differences will be discussed. The comparison will be carried out on the basis of the particular traits of each system. Each comparative criterion will be briefly described, along with the result and difference on each system. A synopsis can be found in table 2.1, which comprises the peculiarities of the two systems.

Major differences according to each criterion:

1. Main processing principle: A major difference of both systems is the conceptual design focus. While Spark focuses on batch processing and treats streaming applications as a special case of micro-batches, Flink follows the opposite philosophy. Flink is mainly focussed on stream processing and simply treats a batch as a finite stream, which is an advantage for real-time requirements.

2. Architecture: Both frameworks implement a master-slave architecture. In Spark the master node is called Spark Driver and the worker nodes are executors. In Flink the terminology for the master node is Job Manager and Task Manager for worker nodes.

3. Programming API: Both systems offer programming interfaces in Scala, Python, Java and SQL. Spark also offers support for R.

4. Libraries: The frameworks include libraries for machine learning, graph processing and SQL. Flink furthermore provides a library for complex event processing (CEP).

5. Available data types: Spark offers RDDs, DataFrames and DataSets while Flink only uses DataStreams (DataSets are depricated). From a developers perspective Flink's data structure is more comprehensive and still serves the major use cases, while in Spark each use case might require a different data structure.

6. Lines of code: Spark is more concise with approximatly 1,200,000 lines of code (LOC) and mainly written in Scala, while Flink consist of 2,100,000 LOC. The figures were measured with gocloc [16] and a detailed table can be found in the appendix (see A.1).

7. Forks on GitHub: Spark has gained a significantly higher popularity with 25,500 forks compared to Flink with 10,600 forks. Furthermore, it can be observed that Flink is discussed noticeably less in online articles as well as scientific contributions.

8. Runtime environment: Both frameworks are executed on Java Virtual Machines.

9. Fault tolerant, in-memory processing: Spark as well as Flink offer fault tolerant execution and in-memory processing.

10. Lazy evaluation: Transformations are performed according to lazy evaluation on both systems. In Spark actions trigger the evaluation chain and in Flink the execute method, which has to be defined in a program, triggers the evaluation.

In conclusion, both frameworks share a structural similarity but differences in the default processing mode and data structures. Furthermore, similar components are named differently in each system. Following this theoretical comparative analysis, an empirical comparison of both systems will be carried out in the application segment (see section 4) based on different hypotheses (see section 3.3).

Criterion	**Spark**	**Flink**
1. Main processing principle	Batch processing	Stream processing
2. Architecture	Master-slave	
3. Programming API	Scala, Python, Java, R, SQL	Scala, Python, Java, SQL
4. Libraries	Machine Learning, Graph processing, SQL	Event processing, Machine Learning, Graph processing, SQL
5. Available data types	RDD, DataFrame, DataSet	DataStream, (DataSet)
6. Lines of code	approx. 1,200,000 primarily Scala	approx. 2,100,000 primarily Java
7. Forks on GitHub	25,500	10,600
8. Runtime environment	JVM	
9. Fault tolerant, in-memory processing	Available	
10. Lazy evaluation	Available	

Table 2.1: Comparative oveview on Apache Spark and Flink (own illustration)

3 Analytical methodology

In preparation for a systematic practical comparison between Apache Spark and Flink, a methodology will be developed in the first step, which will then be applied in chapter 4 by considering different hypotheses.

3.1 Development of the comparison methodology

The practical comparative analysis of both frameworks should, besides a demonstration of the performance of each system, meet the goal of being designed in a reproducible fashion. For this reason, a methodology has been developed prior to the experimental design. It represents a process that breaks down the objective of the elaboration into logical steps and thus systematically answers the underlying problem by means of hypotheses and experimentation. The main objective of this study is to assess the feasibility of deploying weak or heterogeneous hardware within a Spark or Flink cluster.
The adopted methodological approach of the study can be grouped into eight steps, which will be carried out gradually in chapter 4. The major steps of the analyses are illustrated in figure 3.1. At first, the criteria for a comparison of both frameworks will be defined in order to measure the performance of an arbitrary application in a given cluster environment. Based on the intermediate results, hypotheses are formulated in the second step of the analysis. For the empirical verification of the hypotheses a cluster environment as well as a test application is required and the respective requirements will be established in the following steps. The final steps focus on conducting the experiments by first defining a setup, experimentation, interpretation of the results and finally providing an outlook on potential future studies.

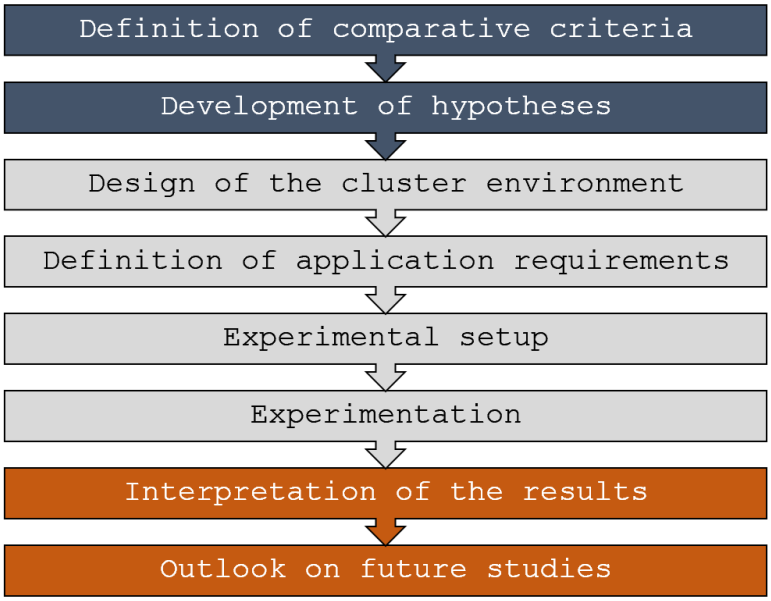

Figure 3.1: Outline of the major steps of the comparative analysis (own illustration)

3.2 Definition of comparative criteria

In order to determine how capable each of the frameworks is in respect to heterogeneous hardware, it is necessary to determine criteria for a structured comparison. First of all, the use cases must be narrowed down to include a reasonable selection of the extensively available APIs of each framework. With the introduction of the theoretical aspects of both frameworks (see chapter 2 for reference) it has been pointed out, that there are basically two main processing modes, namely batch and stream processing. Accordingly, these processing principles will be used for the performance comparison.
Due to the structural differences between the batch and stream processing, different comparison criteria are required. Streaming applications are characterized by the fact, that a certain amount of new data must be processed continuously, whereas batch applications can already access the complete dataset upon startup. Each of the selected performance metrics will be introduced with a brief rationale for the selection.

Performance metrics for comparing batch processing applications:

1. Application lead time: Measures the required time interval between starting a batch application and finishing the job. This metric is an important key process indicator, because it makes the computed data from the calculations available to

other applications earlier as well as it limits the time in which the cluster resources are used. It enables a more efficient exploitation of scarce and costly hardware resources.

2. CPU and RAM utilization: This metric can be used to determine how balanced and efficient the workload is distributed in the cluster.

3. Network bandwidth: The network load can be an indicator for a potential bottleneck in communication between different nodes.

Performance metric for comparing stream processing applications:

1. Throughput: Due to the nature of stream processing, new data is continuously coming into the system at varying amounts per time. This metric measures the amount of data that can be processed per time and thus determines the performance capability of the system.

The introduced comparative criteria provide a basis for the upcoming development of hypotheses, which will include assumptions about the two systems and will be evaluated in the further course of the elaboration of this study.

3.3 Development of hypotheses

The use of heterogeneous hardware in a cluster is a scenario in which the balanced cooperation of the nodes is significantly more complex. The extent to which the performance of a cluster can be affected by different constellations of strong and weak nodes and whether hardware bottlenecks can occur is being investigated. The first hypothesis investigates, if adding a weak Raspberry Pi 3 to a slightly stronger node reduces the overall performance of a two node cluster:

- **Hypothesis 1:** Adding a weak node (Raspberry Pi 3) to another node can decrease the overall performance of the cluster.

Subsequently, an analysis of similar nodes working together will be conducted to measure performance differences between a one and two node cluster. It is assumed, that a significant performance gain can be realized as a result. In this respect the following second hypothesis is being addressed:

- **Hypothesis 2:** Adding a second, similar node can increase the performance of the cluster significantly.

Another interesting aspect of a heterogeneous cluster is the master node, which is responsible for scheduling and deploying applications as well as the management of several worker nodes in a cluster. Due to its relevance in the distributed processing of applications it could cause a potential bottleneck for executing a job. In this regard the third hypothesis will be examined:

- **Hypothesis 3:** The hardware performance of the master node affects the performance of the cluster.

The theoretical comparison between Apache Spark and Flink in section 2.3 has shown that despite similar areas of application of the frameworks, a different philosophy was followed in the design of the systems. In this respect, Spark focuses on batch processing and treats a stream as a series of micro-batches. Flink on the other hand focusses on stream processing and considers a batch as a finite stream. Another major difference is the popularity of both systems. Spark may benefit from more contributions from the community due to a larger user base.
These peculiarities lead to the assumption, that each system is more powerful than their competitor in regard to their favored processing principle. In order to test this systematically, two hypotheses are formulated and examined in the further course of this study by means of experimentation. Due to the focus of this elaboration, the comparison will be concentrated on heterogeneous hardware.
Accordingly, the fourth hypothesis deals with batch processing, assuming that Spark is more powerful for this purpose. The fifth and last hypothesis of this elaboration considers stream processing, assuming that Flink is the more powerful system in this case:

- **Hypothesis 4:** Apache Spark can outperform Apache Flink on batch processing on heterogeneous hardware.
- **Hypothesis 5:** Apache Flink can outperform Apache Spark on stream processing on heterogeneous hardware.

The generated hypotheses will be systematically reviewed in the next chapter and their validity will be tested by means of experimentation.

4 Experiments

Before conducting the experiments for a comparison between Apache Spark and Flink on heterogeneous hardware, the cluster environment and a test application have to be designed. Continuing with planning, conducting and executing the experiments, data is collected and evaluated for the consideration of the hypotheses. In the last step of the experimental part, the results will be discussed.

4.1 Design of the cluster environment

The cluster environment, which shall be used for experimentation, consists of six different nodes and divides in half into stronger and weaker nodes respectively. An overview of the star topology and the configured network is shown in figure 4.1. All nodes are connected via a network cable to a central router (Fritzbox 6591) each with a bandwidth of one gigabit per second.

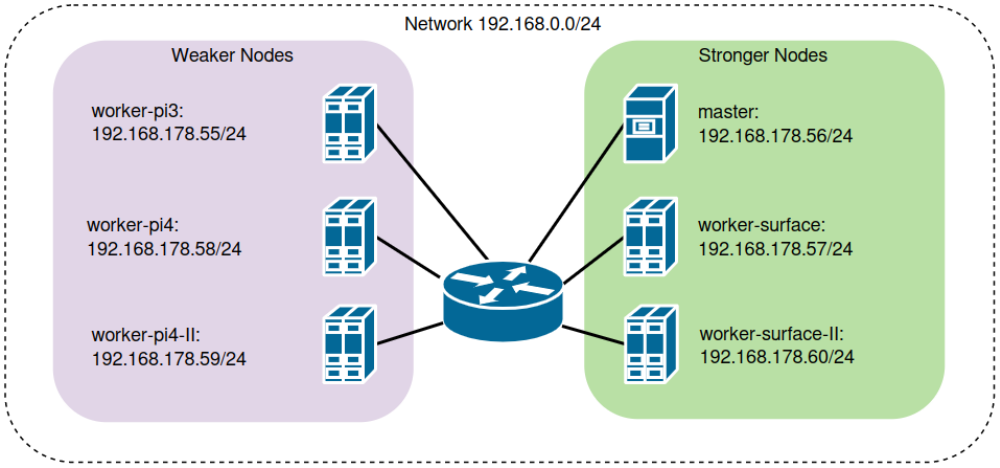

Figure 4.1: The computing cluster consists of 6 nodes which will be used throughout the experiments (own illustration)

Detailed hardware specifications of the deployed nodes are given in table 4.1 which lists name, model, cpu, memory and Ethernet specs for each deployed node.

Name of node	**Model**	**CPU**	Memory	Ethernet
master	Minis Forum PC	4 Cores @ 2,4 Ghz 64-bit AMD 300U	16 GB	1 Gbit
worker-surface	Microsoft Surface Pro 6	4 Cores @ 1,9 Ghz 64-bit Intel i7-8650U	8 GB	1 Gbit
worker-surface-II	Microsoft Surface Pro 6	4 Cores @ 1,6 Ghz 64-bit Intel i5-8250U	8 GB	1 Gbit
worker-pi-3	Raspberry Pi 3 B+	4 Cores @ 1,4 Ghz 64-bit ARM	1 GB	1 Gbit
worker-pi-4	Raspberry Pi 4 B	4 Cores @ 1,5 Ghz 64-bit ARM	4 GB	1 Gbit
worker-pi-4-II	Raspberry Pi 4 B	4 Cores @ 1,5 Ghz 64-bit ARM	8 GB	1 Gbit

Table 4.1: Detailed hardware specification of the given nodes in the cluster

The following software libraries and versions have been used for the Spark cluster:

- Application development: Scala 2.12.15, Spark 3.2.1, sbt[1] 1.5.7 and Java 11
- Cluster environment: Spark 3.2.1 and Java 8
- Deployment: Docker compose version 3 to start the docker containers for the master and slave nodes[2].

The following software libraries and versions have been used for the Flink cluster:

- Application development: Scala 2.12.15, Flink 1.14.0 , sbt 1.5.7 and Java 11
- Cluster environment: Flink 1.14.4 and Java 11
- Deployment: Docker compose version 3 to start the docker containers for the job-manager and taskmanager nodes

[1]Scala build tools, see https://www.scala-sbt.org/ for reference.
[2]For further information see docker-compose.yml file in the projects repository [21].

The described hardware configuration is used both for compiling the test application and for running the cluster. In the following step the test application, which will later be deployed on the cluster, will be designed.

4.2 Definition of test application requirements

To perform the cluster performance tests with Spark and Flink, several applications are required. They serve the purpose of validating the hypotheses as well as measuring the defined performance metrics from section 3.2. The number of required applications depend on the computing frameworks and processing principles.
To benchmark stream and batch processing, separate applications are necessary. In addition, Spark and Flink require different implementations, which results in four test applications being needed. However, the technical implementations between Spark and Flink differ insignificantly. Accordingly, the focus is on the design of the batch and stream application. The implementation details can be found in the project's repository [21].

The following requirements must be satisfied for both batch applications:

- Computational task: The processing of the data should be executable in parallel on different nodes. A word count is to be implemented, that counts the number of occurrences of each key.
- Input Data: The input data for the application is provided in the form of a text file and contains arbitrary keys.
- Output Data: The actual output of the computational task is secondary and can be stored in a textfile. More important are the performance metrics that are collected during the calculations. These should be saved in a .csv file to be used for later evaluation.
- Performance metrics: The application lead time and bandwidth in MBit per second as well as CPU and RAM utilization is being measured. For details on the metrics see section 3.2.
- Programming language: The implementation is written in a native programming language of Spark and Flink, preferably Scala.

- Documentation: The software is documented and commented, with the purpose that developers can interpret the source code and adapt it, if necessary.

The requirements for both stream applications share some traits with the batch applications. The same requirements stated above apply for for the categories output data, programming language and documentation and are to be included in the implementation.

The following requirements must be satisfied for both streaming applications:

- Computational task: The processing application is listening on a socket with a parametrized address for inbound data. A word count is to be implemented, that counts the number of occurrences of each received key.
- Input Data: The input data is provided as a stream of data, the velocity can be parametrized as number of words per second. Another feature is to send as many words in a given time interval. The data is sent to a web socket with a parameterized network address. The definition of the experiment parameters is described in the test setup.

In conclusion, it can be stated that distinct input data must be provided for the applications and processed in the cluster by appropriate calculation procedures. The results of the calculation are relatively negligible and the primary concern lies on the generated performance metrics. This interaction of the system components is illustrated in figure 4.2.

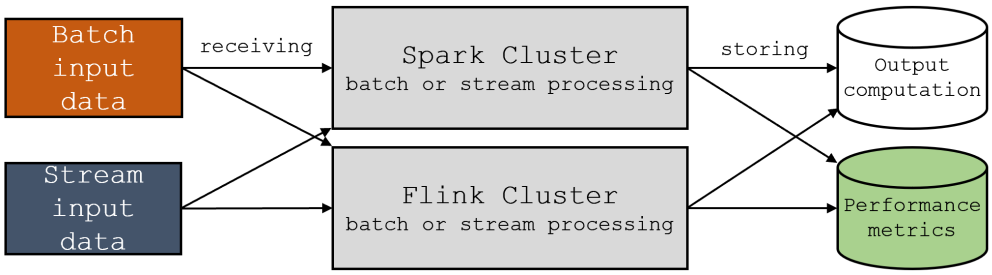

Figure 4.2: The input data has to be processed by both clusters and generates relevant performance metrics for further evaluation (own illustration)

The described requirements for the applications were implemented in the project repository, for further details see [21]. The next step is to prepare the experiments for a

performance test of both systems. For this purpose, the test planning is carried out first.

4.3 Experimental setup

The experimental design that is carried out in this section is based on the cluster environment described in section 4.1 as well as using the applications that were introduced in the previous section. The experimental design is intended to systematically test the hypotheses as well as to ensure reproducibility of the experiments. To cope with measurement inaccuracies, each experiment is performed repeatedly and the calculated mean value is used as the result.
Before analyzing the hypotheses, a duplication test with different input file sizes will be performed. This allows to determine, if the processing time increases linearly with the file size and to obtain a reasonable test file size allowing a reasonable duration for the acquisition of the metrics. The experiment will be conducted on one node (worker-surface) starting with a file size of 25 MiB and doubled each iteration. In order to create the respective files containing a collection of random words with a given size, a python script has been developed[3]. It uses a dictionary of 100.000 distinct words to populate a text file of arbitrary size.

4.3.1 Hypothesis 1

The first hypothesis to be examined claims, that adding a weak node (Raspberry Pi 3) to a more powerful node can have a negative effect on the overall cluster performance. This conjecture is based on the premise that the overhead for synchronizing the progress with a weak node can slow down the process due to its hardware limitations. More specifically, it is expected that longer lead times can be observed when processing an input file after adding the weak node.
The experimental procedure is analogously performed first on the Flink cluster and afterwards on the Spark cluster. The following steps are to be performed for each cluster:

1. Preparing the nodes by booting the Raspberry Pi 3, 4 and the Minis node. The latter is serving as a master node and controlling the others via an ssh connection.

[3]For further information see word_file_creator.py in the projects repository [21].

2. Initiating the monitoring program on all nodes[4]. This python script has been developed to track cpu and memory utilization as well as bandwidth in MBit per second.

3. Startup of the master node followed by the worker nodes in order to connect. Each node is to be started with the respective cluster compute framework.

4. Execution of the batch processing application with a 100MiB input file[5]. This step is repeated for 10 iterations in order to derive a mean value for the runtime of the program.

The generated performance data is stored and documented in section 4.4 and will be interpreted afterwards.

4.3.2 Hypothesis 2

In the course of challenging the second hypothesis, the effect of adding similar nodes to a cluster on the runtime of an application will be analyzed through several experiments. The assumption is, that adding a second similar node will lead to a significant improvement of the runtime of an application. To test this behaviour on the cluster, two sets of experiments will be planned in order to analyze the effect of deploying a second stronger node (Microsoft Surface) on both frameworks and afterwards repeating the process with two weak nodes (Raspberry Pi 4). The following steps will be executed on a Spark as well as on a Flink cluster:

1. Start the respective cluster framework on the Minis node as master.

2. Start the first worker on the Surface node and connect to the master node.

3. Execution of the batch processing application with an input file with 500 MiB. This step is repeated for 10 iterations in order to derive a mean value for the runtime of the program.

4. Prepare the second worker node (surface-II) and connect to the master node.

5. Repeat step 3, now with two connected worker nodes.

[4]See node_monitoring.py in the projects repository [21].
[5]See flink-word-count_2.12-1.0.jar in the projects repository [21].

After finishing the documented steps above, the same procedure is to be repeated with Raspberry Pi 4 nodes instead. The results of the experiments shall later be used to evaluate the second hypothesis.

4.3.3 Hypothesis 3

The master node will be the focus of the third hypothesis. More specifically, experiments will be conducted to measure the influence of the hardware capabilities of the master node on the performance of the cluster. In this regard, two cluster setups will be compared. First, using a strong master node and a strong worker node and afterwards exchanging the master node by a weaker node to see the effect on the runtime of an application. The following steps will be executed on a Spark cluster:

1. Preparing the master node by booting the Minis node and starting the Spark Master process.
2. Starting the worker process on the Surface node and connect it to the master node.
3. Execution of the batch processing application with a 100MiB input file[6]. This step is repeated for 10 iterations in order to derive a mean value for the runtime of the program.
4. Repeat step 1 with the Raspberry Pi 4 as a master node.
5. Repeat steps 2 and 3.

The generated performance data will be used to challenge the third hypothesis.

4.3.4 Hypothesis 4

The fourth hypothesis to be examined assumes, that Apache Spark can outperform Apache Flink on batch processing on heterogeneous hardware. This conjecture is based on the premise that Spark's specialization in batch processing has a positive impact on its performance compared to Apache Flink. More specifically, it is expected that shorter lead times can be observed when processing an input file.
In order to test this behaviour, several batch processing applications have to be deployed on a Spark and Flink cluster and the results have to be compared. For this analysis

[6] spark-word-count_2.12-1.0.jar in the projects repository [21].

the data from the experiments of hypothesis 2 can be leveraged. It provides a basis for different one and two node cluster setups of both frameworks.
Furthermore, another experiment will be performed to compare the performance of an additional cluster setting. In order to do so, the same procedure of the experiments of the second hypothesis will be repeated with a three node cluster. In this scenario an input file will be processed using a single surface worker node in the first step. Afterwards, two Raspberry Pi 4's will successively be added to the cluster and the same input file is being processed.

4.3.5 Hypothesis 5

The last hypothesis to be investigated in the course of this study postulates, that Apache Flink can outperform Apache Spark on stream processing on heterogeneous hardware. Accordingly, different cluster setups with one or two worker nodes will be deployed to measure the performance of both frameworks. The following steps will be performed on Spark and Flink respectively:

1. Open a TCP socket on port 9000, which will later be used to send a stream of words. This can be achieved by using a python script[7], which has been developed for this project.

2. Start the master node as well as start and connect the Surface worker node.

3. Deploy the streaming application[8] on the cluster, which is connecting to port 9000 and processing the data by word count. The processing window is configured with an interval of 1 second.

4. Specify the experiment duration to 10 seconds in a prompt of the python script. The maximum throughput of words will be measured during the time interval.

5. Repeat step 1 to 4 with the Raspberry Pi 4 as worker node and afterwards use both nodes together.

The conducted experiments for both cluster frameworks will be yielding the amount of words processed during each iteration. The average throughput per second will be

[7] word-stream-creator.py in the projects repository [21].
[8] WordCountStream class in flink-word-count_2.12-1.0.jar or WordCount class in spark-word-count_2.12-1.0.jar respectively in [21].

used to compare the performance of both frameworks for stream processing and thereby generating data to examine the fifth hypothesis.

4.4 Experimental results and interpretation

This chapter documents the results of the experiments that were performed. The tests were carried out on the basis of the experimental setup (see section 4.3) and will be interpreted after each documentation of the results.
Before conducting the experiments for the different hypotheses, a doubling test has been carried out first. The results of the experiment on one Flink node are illustrated in figure 4.3 yielding a linear influence of the file size on the runtime of a batch program. Starting with 25 MiB the file size has been doubled four times up to 400 MiB. The runtime increased from 3.46 sec to 6.66 sec (92.49 % increase) for the first doubling of the input file size from 25 to 50 MiB. The last iteration on 400 MiB took 46.74 sec. On average a doubling increased the runtime by 92.71 %.
On the basis of this experiment all further tests can be done with a file size that leads to a well testable experiment duration, since the effect on the runtime is linear. Therefore the duration of an experiment will be chosen with a reasonable length (by the size of the input file) to accommodate variations in processing speed as well as an appropriate window of time to measure the respective performance metrics.

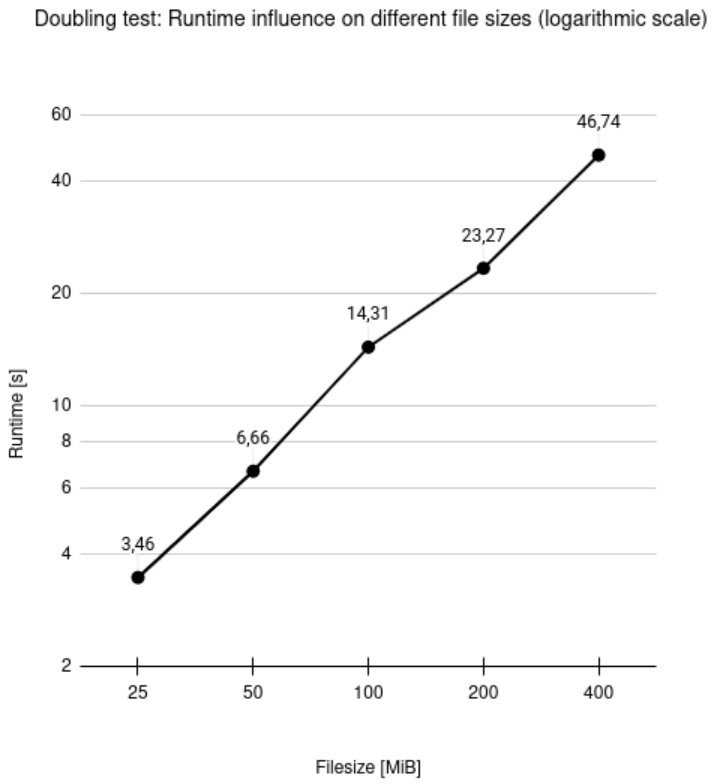

Figure 4.3: The file size has a linear influence on the runtime (own illustration)

4.4.1 Hypothesis 1

The experiment for testing the first hypothesis was conducted with two nodes, first running solely on a Raspberry Pi 4 and afterwards adding a Raspberry Pi 3 with limited hardware capabilities.
When using a single Flink node for batch processing a 100 MiB file containing random words in stream execution mode the average runtime was 37.38 sec. Processing the same bounded file with Flink in batch execution mode yielded a higher runtime of 115.12 sec and sometimes causing application failures on the worker node caused by 'out of memory' errors. This behaviour occurs when the experiment is repeated several times, each time leading to an increasing amount of memory used on the node. The corresponding monitoring data for this case is shown in figure A.3 in the appendix. Processing the file with a single Spark node yielded an average runtime of 30.1 sec.
After adding the second node (Pi 3) a negative effect on the runtime has been observed

yielding a lead time of 43.07 sec (15.2 % performance loss) in Flink's stream processing mode. In contrast, no significant effect on the runtime was observed for processing the file in batch mode when executed on both Flink nodes. The lead time yielded 116.87 sec. A negative effect was observed in terms of stability, both nodes tended to crash more often when combined. Adding the Pi 3 on Spark could not be tested. The job execution fails due to resource problems after submitting the application to the cluster. The error log indicates, that the worker doesn't have sufficient memory resources. The results of the experiment are illustrated in figure 4.4.

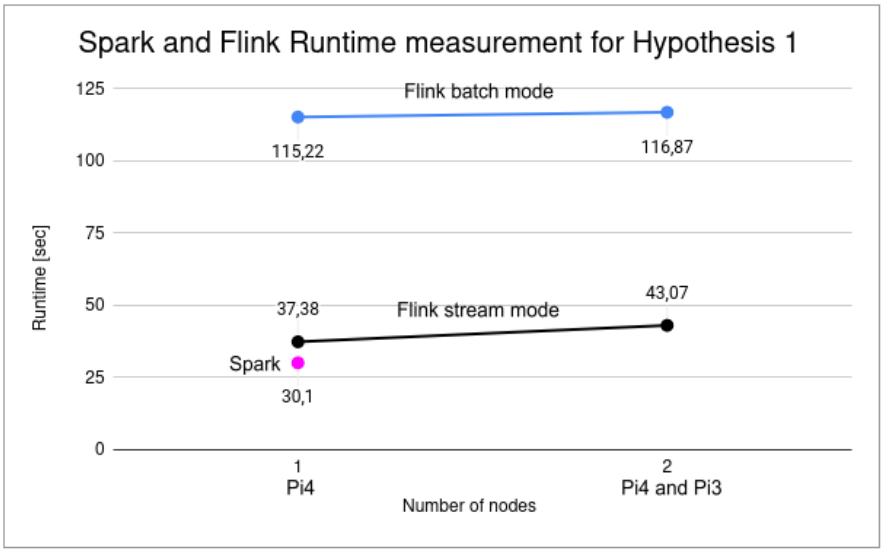

Figure 4.4: Raspberry Pi3 shows negative effect on runtime when added to cluster (own illustration)

The results of the hardware monitoring of the Flink cluster is described subsequently, covering the execution of the batch task (in stream execution mode) on the Flink cluster comprising three nodes. First, the cpu usage is analyzed, followed by the memory usage and finally the observed network traffic on the three nodes.
The cpu monitoring of the nodes deployed on the Flink cluster in figure 4.5 is showing the time span from the start of the cluster, the actual experimentation and the idle cluster after the experiment. It shows, that the cpu utilization on the Raspberry Pi 3 is relatively high at a maximum of around 95 % during the experiment. This observation can indicate a bottleneck on this node. In contrast, the load on the Pi 4 is significantly lower. The Minis node is not working to full capacity and reaching a maximum of 20 %

during the experiment. When the cluster was idle after finishing the job, the observed cpu load was less than 10 %.

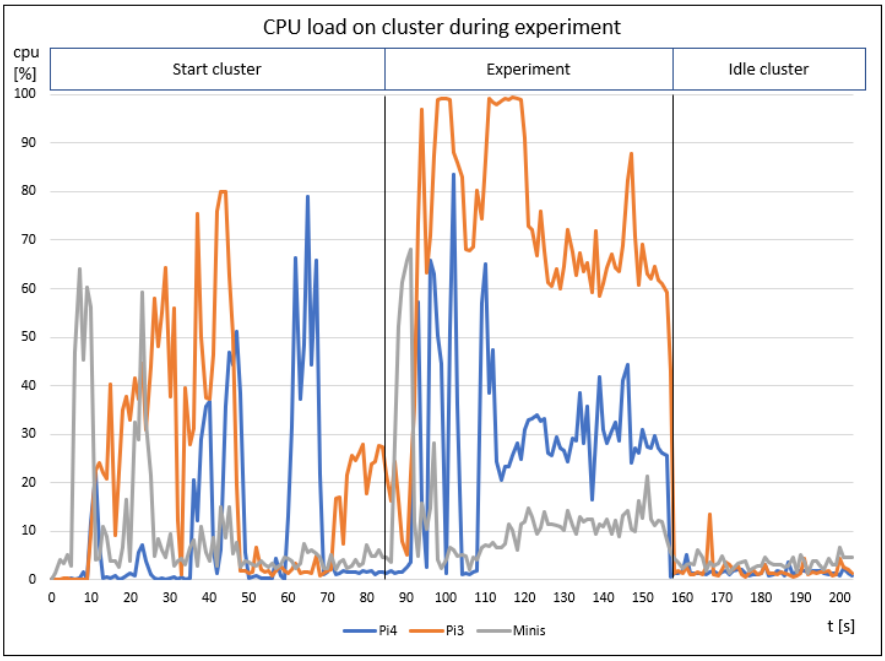

Figure 4.5: CPU monitoring on worker and master nodes during experiment (own illustration)

The memory monitoring of the nodes in figure 4.6 shows, that the memory usage on the Raspberry Pi 3 is relatively high and using up to 76 % which might indicate a bottleneck. In contrast, around a maximum of 40 % have been used on the Pi 4 and around 20 % on the Master Node (Minis PC). Another interesting effect is, that after the experiments the memory is still allocated on the nodes and not returning to its initial level as before the experiment.

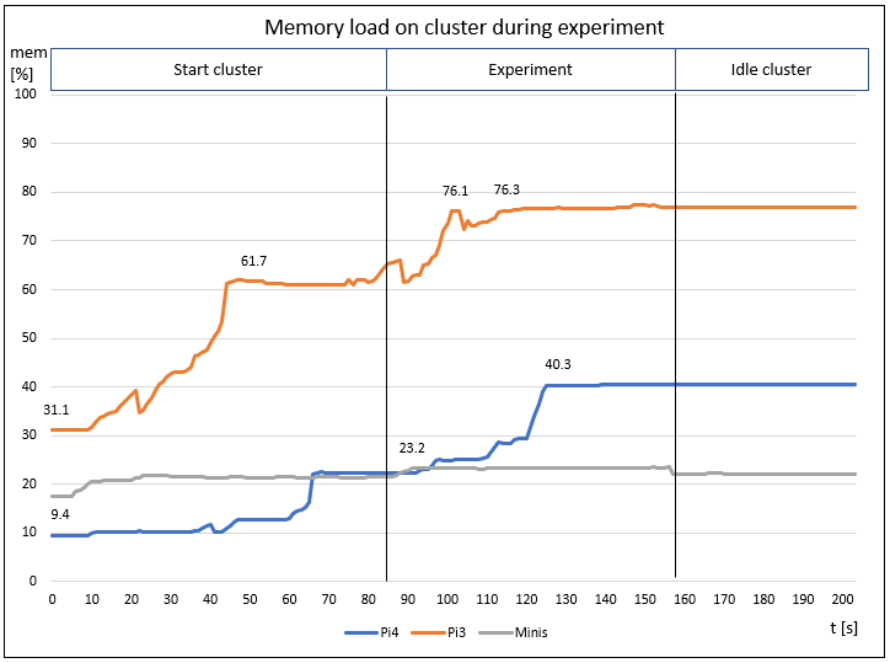

Figure 4.6: Raspberry Pi3 memory resources are heavily utilized (own illustration)

The monitoring of the network activity of the nodes is illustrated in figure 4.7. During the start of the cluster an initial peek can be observed on the master node which might be caused by establishing a connection to the worker nodes as well as the web interface of the cluster. During the experiment a similar load can be observed on both worker nodes between 20 and 25 MBit per second. This load is likely to be caused by the synchronization between both worker nodes on the shared task of processing the input file.

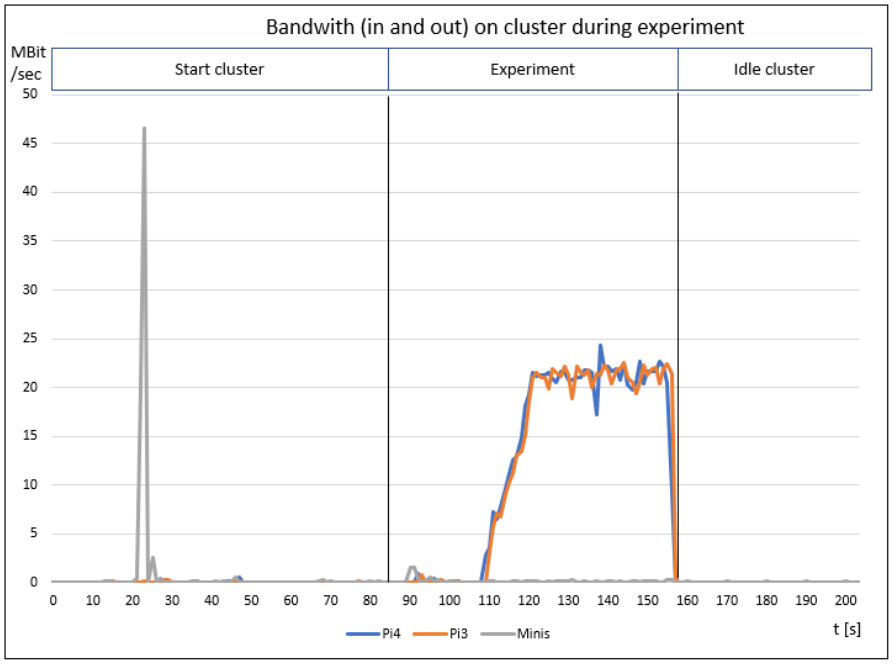

Figure 4.7: Network monitoring on worker and master nodes during experiment (own illustration)

In conclusion, the hypothesis is supported by the experiments. The addition of the weak node (Raspberry Pi 3) has reduced the overall performance of the cluster. When deployed on the Flink cluster a performance loss of around 15 % has been measured (in stream execution mode). The monitoring showed a cpu and memory bottleneck on the Pi 3 which might limit the performance of the cluster. Another potential bottleneck could be the network bandwidth. In order to verify this bottleneck the bandwidth would need to be compared to a stronger node. When deploying the Pi 3 node to the Spark cluster the execution fails and thus supports the hypothesis. A possible explanation are the limited hardware resources, which are not sufficient to handle the cluster overhead as well as the application.

Another interesting finding is, that processing a bounded file in batch mode was slower (around 200 % increase in runtime) and less stable than executing the same file in stream mode. This was unexpected, especially since the official Flink documentation states processing bounded inputs in batch modes is more efficient[9].

[9]See documentation for Flink processing modes for reference: https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/dev/datastream/execution_mode/, last visited 13.06.22.

4.4.2 Hypothesis 2

The experiments in this section are measuring the effect of adding a second worker node with similar hardware specs on the runtime of applications on a cluster. The corresponding second hypothesis assumes, that adding a second node will significantly improve the average runtime on both Spark and Flink. Therefore two distinct sets of experiments have been laid out. First, by testing the effect of deploying a second Microsoft Surface Node on both frameworks. Afterwards the same will be tested with two Raspberry Pi 4 worker nodes, which are considerably slower than the first set of nodes.
The results of the first set of experiments are illustrated in figure 4.8 and are tested on two Microsoft Surface nodes to measure the mean processing time of a 500 MiB file containing random words. Upon using a single Spark worker node a runtime of 16.3 sec has been observed. After adding a second node the runtime has been shorter at 12.5 sec, yielding a 23.31 % improvement.
Using Flink with a single worker node resulted in a runtime of 45.77 sec by processing the file in stream mode. Adding a second node lead to a better runtime of 25.78 sec with an improvement by 43.67 %. The observed runtime for processing the file in batch execution mode on Flink showed a much slower runtime of 87.21 and 76.84 sec respectively. This observation is unexpected, due to the fact that batch execution is considered to be more efficient as stated in the Flink documentation [7]. One assumption is, that Flink is not optimized for processing files due to it's streaming focus. Another possible cause might be the examined scenario on heterogeneous hardware. It is different to the usual application on highly capable hardware of computing centers, which can contain hundreds of nodes.

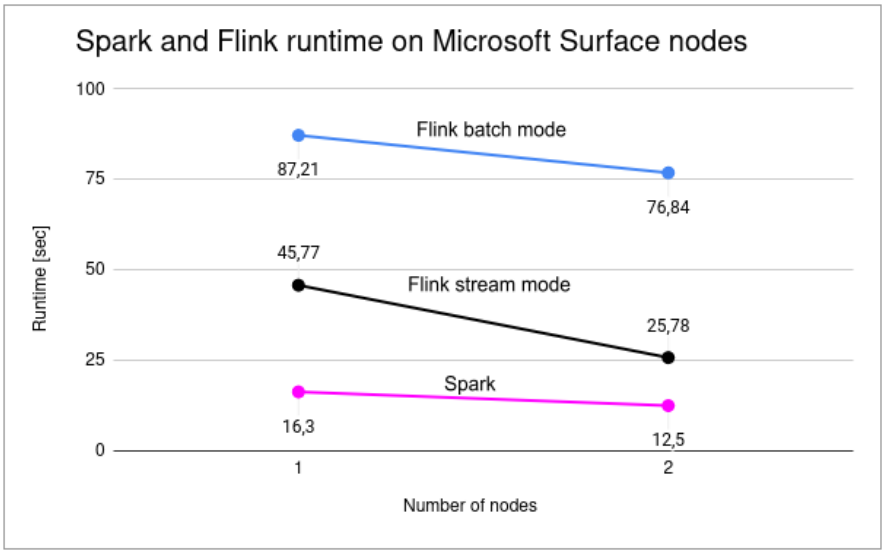

Figure 4.8: Runtime improvement by deploying a second Microsoft Surface node using Spark and Flink (own illustration)

The results of the second set of experiments are illustrated in figure 4.9 and are tested on two Raspberry Pi nodes to measure the mean processing time of a 100 MiB file containing random words. When using a single Spark worker node a runtime of 35.7 sec has been observed. After adding a second node the runtime has been lower at 27.1 sec, yielding a 24.09 % improvement.
Using Flink with a single worker node resulted in a runtime of 32.42 sec by processing the file in stream mode. Adding a second node lead to a shorter runtime of 18.5 sec with an improvement by 42.94 %. The observed runtime for processing the file in batch execution mode on Flink showed a slower runtime of 94.42 and 72.11 sec respectively. The effect of a slower runtime for batch instead of stream processing on Flink has been observed again.

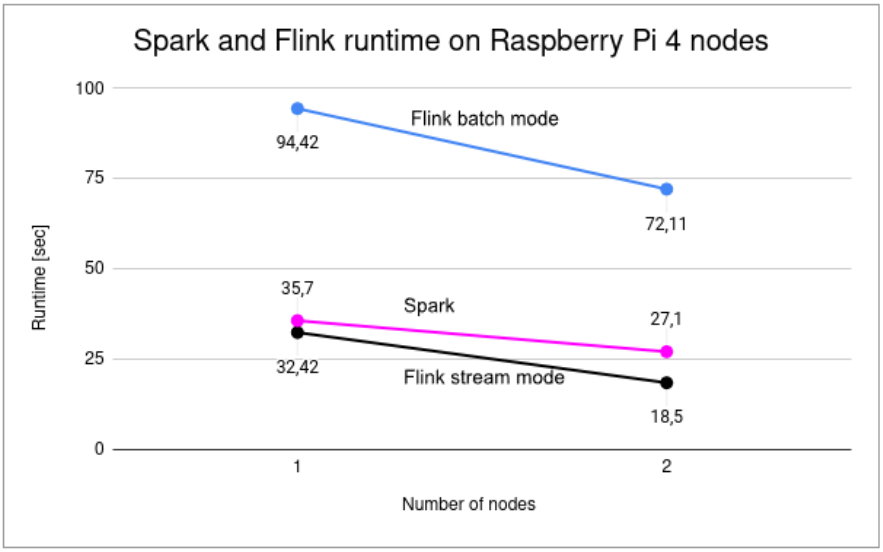

Figure 4.9: Runtime improvement by deploying a second Pi 4 node using Spark and Flink (own illustration)

Both sets of experiments support the statement of the second hypothesis. Adding a second, similar node increased the performance of the cluster significantly in the tested scenarios. The runtime has been improved both on stronger Microsoft Surface nodes and weaker Raspberry Pi 4 nodes with both frameworks. The observed relative improvement of adding a second node was higher on Flink. Flink improved by 43.31 % on average, Spark on the other hand improved by 23.70 %.

4.4.3 Hypothesis 3

The experiments for testing different master nodes follow the goal to measure the influence of the hardware capabilities of the master node on the performance of the cluster by benchmarking the application lead time. In order to do so, two experiments have been conducted, each in a two node setup. First, by deploying a stronger Spark master node (Minis) with a worker node (Microsoft Surface) and afterwards repeating the experiment with a weak Spark master node (Pi 4) with the same worker node.
The runtime results upon using the stronger master node yielded a runtime of 9.2 sec for batch processing a 100 MiB input file on the single worker node (see figure 4.10). When deploying the weaker master node an average runtime of 13.7 sec has been observed. In comparison, a 48.91 % slower application lead time has been observed when using the

inferior node in terms of hardware capabilities.

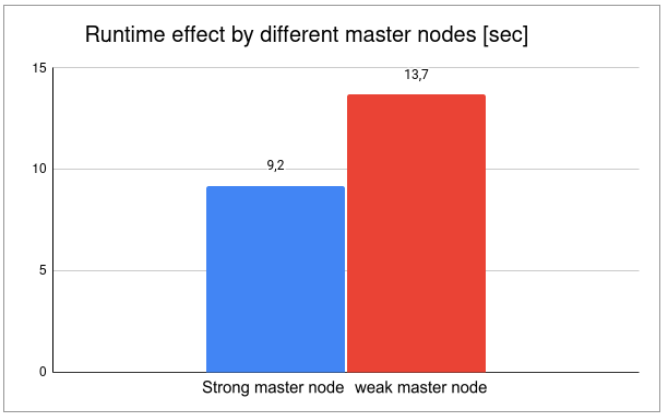

Figure 4.10: Runtime is affected by hardware performance of master node (own illustration)

Due to the observed increase in runtime, caused by inferior hardware resources a further analysis of a hardware bottleneck has been performed by monitoring the cpu, memory and network utilization during both experiments. In the course of the performance monitoring a higher cpu load was detected on the Pi 4, utilizing up to 83.4 % of available cpu resources as illustrated in figure 4.11. The load on the stronger node was generally lower with a short peak of 73.8 % cpu load. Furthermore, the relative load needs to be put in context to the cpu clock frequency of each node. The clock frequency of the Pi 4 offers 1.5 Ghz and is slower than the Minis node with 2.4 Ghz and likely causing the runtime difference. The observed network traffic and memory utilization does not indicate to cause the bottleneck on runtime. Both nodes had sufficient free memory and just showed a short period of network traffic. For further details on these metrics, see appendix A.3.

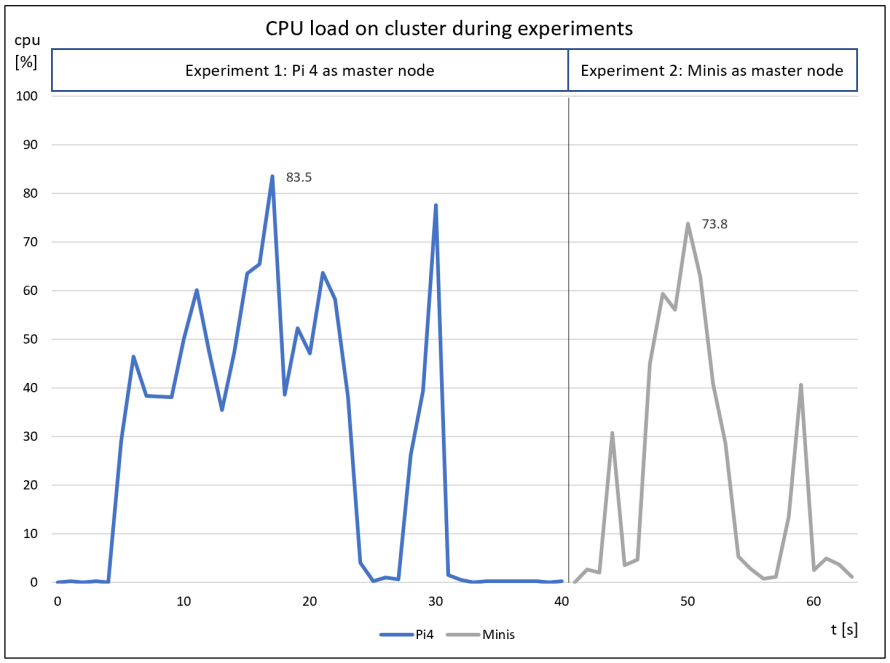

Figure 4.11: Cpu monitoring showing a higher utilization on the Pi 4 than the Minis node (own illustration)

In conclusion, the third hypothesis can be supported by the observations of both experiments. The hardware performance of the master node showed an effect on the performance of the cluster. In comparison, the deployment of a weaker Spark master node lead to a slower application lead time by around 49 %. Therefore, the deployment of a strong master node can be recommended in a heterogeneous cluster environment to avoid a potential bottleneck of the cluster performance.

4.4.4 Hypothesis 4

It was postulated, that Apache Spark can outperform Apache Flink on batch processing on heterogeneous hardware. In order to verify this assumption, the data of the experiments of the second hypothesis will be analyzed as well as further experiments for comparing both frameworks on a three node cluster.
The results of the runtime of processing a 500 MiB file on one and two Microsoft Surface nodes has been introduced in figure 4.8. It showed, that Spark had a faster average runtime of 14.4 sec. compared to Flink with 35.78 sec. Flink needed around 2.5 times as long as Spark for processing the file.

When deployed on one or two Raspberry Pis (see figure 4.9) to process a 100 MiB file Spark had a slower average runtime of 31.4 sec in comparison to Flink with 25.46 sec. Spark needed around 1.23 times as long as Flink for processing the file.

The results of the additional experiments are illustrated in figure 4.12 and are tested on three worker nodes (Microsoft Surface and two Raspberry Pi 4) to measure the mean processing time of a 100 MiB file containing random words. Upon using a single Spark worker node (Microsoft Surface) a runtime of 8.9 sec has been observed. After adding a second node the runtime is roughly the same at 9 sec. The node added in the third run also just had a minor negative impact and yielded a runtime of 9.1 sec.
Using Flink with a single worker node resulted in a runtime of 11.4 sec for processing the file in stream mode. Adding a second node lead to slower runtime of 25.92 sec with an deterioration by 127.37 %. Adding a third node led to a runtime of 15.06 sec, which is still slower than running solely on a single stronger node. The observed runtime for processing the file in batch execution mode on Flink showed a slower runtime of 22.86, 47.27 and 35.11 sec respectively. In general, Spark has been faster in any of the tested scenarios and also showed a lower deterioration after adding the two Pi 4 nodes.

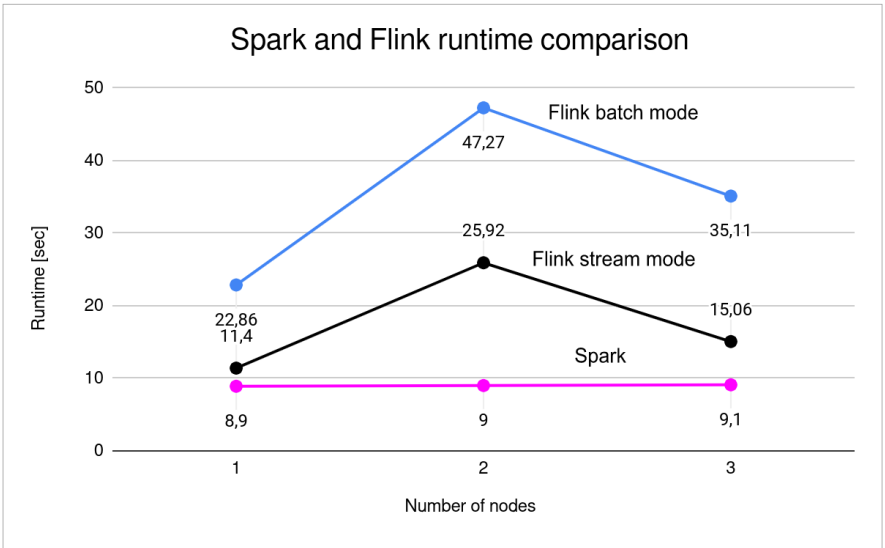

Figure 4.12: Shorter runtime observed on Spark three node cluster. Weak nodes decreased performance of a strong node (own illustration)

In general Spark has been faster in all but one case. Only when deployed on two Raspberry Pis it showed lower performance for processing bounded files. But this can be

considered as a special case, because the cause for the difference is most likely due to the different memory management. Flink allows different memory settings per node whereas Spark uses the same memory allocation on all executor nodes. This means, that in the tested scenario where Spark was slower it had less memory to operate on. A further problem is that in that special case it was only faster in stream mode and the focus of the fourth hypothesis is on processing in batch mode. Therefore the hypothesis was not refuted and several scenarios were found in which Spark outperformed Flink on batch processing.
Another interesting finding is the effect of adding weak nodes (Raspberry Pi 4) to a stronger node (Microsoft Surface) which caused a decrease in runtime performance on the overall cluster. It can be concluded, that a certain level of hardware capabilities is required for a reasonable deployment in a bigger cluster setting. Nodes similar to the performance of Raspberry Pis had shown limited benefit to a cluster system. Furthermore it can be recommended, that the nodes should be equipped with similar or equal memory size. This simplifies the configuration of the cluster significantly and avoids potentially idle resources.

4.4.5 Hypothesis 5

The experiments for challenging the last hypothesis focus on the capabilities of Apache Spark and Flink on stream processing. Therefore, three experiments have been conducted to measure the throughput of both frameworks in different cluster settings.
The results of the experiments are shown in figure 4.13 and are tested on two worker nodes (Microsoft Surface and Raspberry Pi 4). First, each node has been tested individually and then combined.
When testing a single, stronger Spark node (Surface) an average throughput of 280,601 words per second has been observed. The same experiment with the Raspberry Pi yielded a lower throughput of 266,531 words per second. Both nodes combined generated a lower performance than a single stronger node with a 268,666 words per second. For the set of experiments on the Flink cluster 295,045 words per second have been measured on the Surface node as well as 268,323 and 284,983 words per second for the Pi 4 and both nodes together.

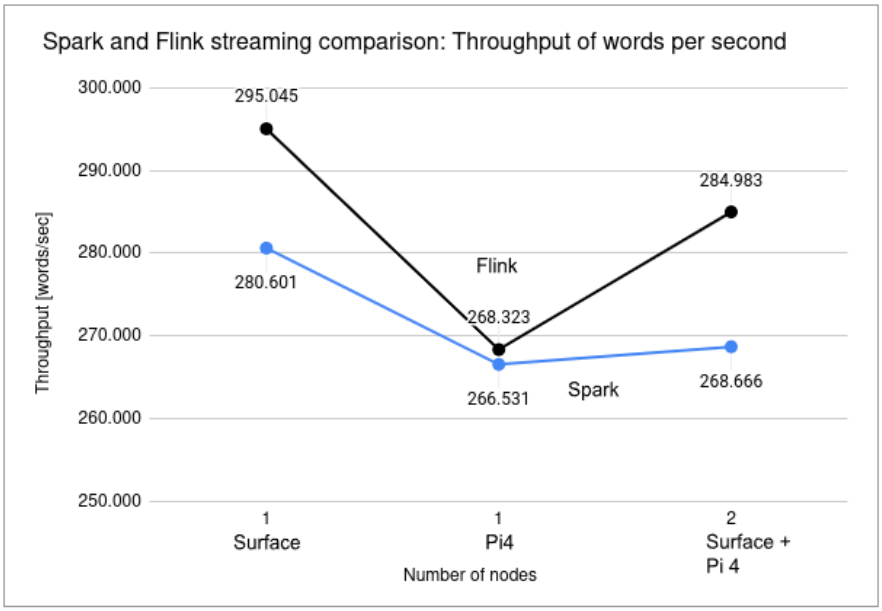

Figure 4.13: Apache Flink showed better performance on stream processing (own illustration)

Overall, Flink demonstrated a better performance in all three scenarios for stream processing words via TCP streams. On average of all experiments the performance of Flink has been around 4 % higher than on Spark. In conclusion, the fifth hypothesis can be supported by the observations. In the tested scenario the Flink framework can be recommended instead of using Spark. Another qualitative difference was a more flexible API provided by Flink. It offers a variety of configurations for streaming applications as well as different notions of time and out of order functionality for stream processing. Furthermore, the experiments showed that the Raspberry Pi 4 has decreased the overall cluster performance when combined with stronger nodes. In the last section, the same phenomenon was already observed for batch processing.

4.5 Discussion

In the course of this project, various experiences with both frameworks were gained by conducting experiments and evaluating the hypotheses. The most notable experiences and the consideration of expectations will be discussed here.
A key learning has been that both frameworks managed to operate on all systems, despite

weak hardware. However, no standard docker images could be utilized on a Raspberry Pi, but instead custom Docker files had to be developed that included customized versions for the ARM chip. Furthermore, proper versioning of the used tools (e.g. JVM, Scala, Spark or Flink version) was crucial for a flawless cooperation of the nodes during parallel execution. Minor version deviations within the cluster caused compatibility problems that could not be interpreted by error logs and resulted in a considerable amount of development time being spent for debugging.
Another experience represents the configuration of a cluster. There is a multitude of configuration parameters for a cluster, e.g. the number of executors per node or memory allocation. Therefore fair benchmark comparisons become more challenging to realize. In this sense, published benchmark results have to be considered critically, to assure the underlying use cases and configurations have been chosen fairly for the respective systems. Consequently, an extensive documentation of the adopted configuration has been done throughout this project and in addition, the cluster configurations have been published within the GitHub repository.
When setting up a cluster with the respective framework, a careful network configuration is necessary. This ensures accessibility between all nodes and is essential for the functionality of the cluster.
For the development of a test environment sufficient time must be allocated accordingly. In practice, Flink showed a modest advantage in terms of the time required for a setup. The support provided by the Flink framework with its prepared configuration files was more efficient than Spark in this respect.
In general, sound Linux knowledge is recommended when attempting to use different cluster configurations, as it requires frequent connections and modifications of nodes. Remote control of the nodes in combination with automation by using shell scripts can save a considerable amount of time. The scripts, that have been used for this purpose can be found in the project's repository. In contrast, this is probably less relevant in the day-to-day deployment on a datacenter, where a vast number of nodes are provided and only the .jar file (and possibly input data) needs to be deployed.
Last but not least, a higher popularity of a framework simplifies debugging by harnessing documented problems of developers by following published discussions on the internet. For instance, much more information was available for Spark. With Flink, on the other hand, there is less discussion and the framework's documentation is the main source for troubleshooting, such as correct configurations in a multi-node cluster.
It was anticipated, that Spark would perform better on batch processing but it was rather unexpected, that such a major difference could be observed (Flink needed up to

2.5 times as long as Spark). A certain performance advantage may be explained by a different design focus of both frameworks on different processing principles, but not in this order of magnitude. Furthermore, it has not been adequately explained, why Flink is slower at processing files in batch mode than in stream mode. A possible explanation was the design focus of Flink or the uncommon hardware that was used.
In general, it was expected that weak nodes, represented by Raspberry Pis, would be significantly less powerful. However, it was surprising that they decreased the performance of the system in combination with stronger nodes. This bottleneck appears to be induced by insufficient hardware resources.

In conclusion, the practical comparison of both frameworks in this particular context with heterogeneous hardware has provided a set of valuable insights and future research could benefit from the acquired learnings. Subsequent fields of research will be discussed in the following, final chapter.

5 Summary and future work

Several insights for deploying Apache Spark and Flink on heterogeneous hardware in a small cluster environment have been obtained in the course of this elaboration. The essential findings, along with an outlook on further research will be given in this section.

5.1 Summary

Initially, the theoretical aspects of Spark and Flink along with a comparison of architectures, data structures, and libraries were given as a basis to develop a cluster environment. Subsequently, an analysis methodology was designed to compare both frameworks, which has been applied later in the experimentation section.
In the context of the objective to investigate both frameworks for the use with heterogeneous hardware, five hypotheses were developed, which were confirmed by the conducted experiments. The first three deal with the influence of different hardware on the cluster's performance in varying combinations of nodes working together. The remaining two compare the performance of both frameworks in regard to batch and stream processing. Following the definition of a cluster environment, the development of test applications, and the design and setup of the individual experiments, performance data was collected by performing the experiments. The following findings have been derived as a result of challenging the hypothesis by experimentation:

- It was observed, that adding weak nodes to a stronger node reduced the overall performance of the cluster.
- Another finding is, that the combination of similarly powerful nodes leads to an increase in the performance of the overall system.
- The performance of the master node has a significant impact on the performance of the cluster. In general, it can be recommended to deploy a strong master node in order to avoid a potential bottleneck.

- Depending on the processing principle, performance differences between both frameworks have been identified. In the case of batch processing, Apache Spark performed significantly better, while Flink was the superior system for stream processing.

In general, weak nodes such as Raspberry Pi 3 and 4 are not recommendable to be used in a cluster environment for big data processing. Due to the fact, that they decreased the performance of the cluster when combined with stronger nodes. Promising results have been obtained when deploying stronger nodes on Spark and Flink cluster environments.

5.2 Outlook

In the course of this study, additional research questions were developed that might be of interest as a sequel. Since the deployment of stronger, yet heterogeneous nodes showed promising results, it would be interesting to test a multi node cluster environment. The gathered performance benchmarks could then be compared to commercial computer centers and considered in the perspective of acquisition and maintenance costs. Another configuration parameter which has not been investigated, is the number of executors or task slots per node and the influence on the cluster performance. Especially for Spark, this could be an approach, to fully utilize the working memory of each node since all executors must have the same memory space for distributed processing. It has been pointed out, that Raspberry Pis have been to weak to be deployed on a cluster. To test them in a multi node cluster might be a scenario, in which there could be a threshold for competing with stronger nodes. For comparing batch and stream processing the scope of this elaboration was limited to a single use case for each application. It would be worthwhile, to test additional use cases and compare them to the results of this study. In commercial applications the integration of distributed databases or datalakes as well as event streaming platforms like Kafka would be relevant for a performance comparison as well. Another potential use case could be comparing the machine learning libraries offered by the frameworks (Mlib and FlinkMl). Furthermore, the application of near real time streaming could be an interesting follow-up study. The latency of each system could be used as an additional performance metric for comparison. Lastly, a comparison of Spark and Flink with further compute frameworks, e.g. like Apache Storm or Samza might be promising to investigate, whether there are more efficient frameworks available in the growing technological landscape for Big Data processing.

Bibliography

[1] Agneeswaran, Vijay S. (Hrsg.): *Big Data Analytics beyond Hadoop*. First edition. Pearson Education, 2014

[2] Bifold: *Apache Flink - From academia into the apache software foundation and beyond*. 2022. – URL https://bifold.berlin/de/tag/data-management/. – Zugriffsdatum: 31.03.2022

[3] Carbone, Paris ; Ewen, Stephan ; Fóra, Gyula ; Haridi, Seif ; Richter, Stefan ; Tzoumas, Kostas: State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. In: *Proc. VLDB Endow.* 10 (2017), aug, Nr. 12, S. 1718–1729. – URL https://doi.org/10.14778/3137765.3137777. – ISSN 2150-8097

[4] Chellappan, Subhashini (Hrsg.): *Practical Apache Spark – Using the Scala API*. First edition. Apress, 2018

[5] DataFlair: *Spark RDD Operations: Transpformations and Actions*. 2022. – URL https://data-flair.training/blogs/spark-rdd-operations-transformations-actions. – Zugriffsdatum: 21.03.2022

[6] Foundation, Apache S.: *Apache Flink - Stateful Computations over Data Streams*. 2022. – URL https://flink.apache.org. – Zugriffsdatum: 31.03.2022

[7] Foundation, Apache S.: *Apache Flink Documentation*. 2022. – URL https://nightlies.apache.org/flink/flink-docs-release-1.14. – Zugriffsdatum: 31.03.2022

[8] Foundation, Apache S.: *Apache Flink ML repository*. 2022. – URL https://github.com/apache/flink-ml. – Zugriffsdatum: 19.04.2022

[9] Foundation, Apache S.: *Apache Flink repository*. 2022. – URL https://github.com/apache/flink. – Zugriffsdatum: 31.03.2022

[10] Foundation, Apache S.: *Apache Spark Cluster Overview*. 2022. – URL https://spark.apache.org/docs/latest/cluster-overview.html. – Zugriffsdatum: 16.03.2022

[11] Foundation, Apache S.: *Apache Spark history*. 2022. – URL https://spark.apache.org/history.html. – Zugriffsdatum: 15.03.2022

[12] Foundation, Apache S.: *Apache Spark official documentation*. 2022. – URL https://spark.apache.org/docs/latest/. – Zugriffsdatum: 18.03.2022

[13] Foundation, Apache S.: *Apache Spark repository*. 2022. – URL https://github.com/apache/spark. – Zugriffsdatum: 15.03.2022

[14] Freiknecht, Jonas (Hrsg.): *Big Data in der Praxis - Lösungen mit Hadoop, Spark, HBase und Hive*. Second edition. Hanser, 2018

[15] García-Gil, Diego ; Ramírez-Gallego, Sergio ; García, Salvador ; Herrera, Francisco: A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. In: *Big Data Analytics* 2 (2017), Nr. 1, S. 1–11

[16] Hattori, Hideo: *gocloc GitHub Repository*. 2022. – URL https://github.com/hhatto/gocloc. – Zugriffsdatum: 22.04.2022

[17] Hueske, Fabian (Hrsg.): *Stream Processing with Apache Flink - Fundamentals, Implementation, and Operation of Streaming Applications*. First edition. O'Reilly, 2019

[18] Kaepke, Marc (Hrsg.): *Graphen im Big Data Umfeld - Experimenteller Vergleich von Apache Flink und Apache Spark*. Hochschule angewandte Wissenschaften Hamburg, 2017

[19] Karau, Holden (Hrsg.): *Learning Spark - Lightning-fast data analysis*. First edition. O'Reilly, 2015

[20] Krettek, Aljoschka: *Consolidate the user-facing Dataflow SKDs and APIs*. 2020. – URL https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=158866741. – Zugriffsdatum: 20.04.2022

[21] Könemann, Alexander: *thesis-spark-vs-flink-koenemann*. 2022. – URL https://github.com/alexk314/thesis-spark-vs-flink-koenemann. – Zugriffsdatum: 15.06.2022

[22] Marcu, Ovidiu-Cristian ; Costan, Alexandru ; Antoniu, Gabriel ; Pérez-Hernández, María S.: Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, S. 433–442

[23] Masurat, Finn (Hrsg.): *Realtime-Erkennung von Hate-Speech auf Twitter: Eine Evaluation von Apache Flink und Spark*. Hochschule angewandte Wissenschaften Hamburg, 2019

[24] Santos, Wendell: *FlinkMl 2.0 released with updated API and Python SDK*. 2022. – URL https://www.programmableweb.com/news/flinkml-20-released-updated-api-and-python-sdk/brief/2022/01/14. – Zugriffsdatum: 19.04.2022

[25] Statista: *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025*. 2022. – URL https://www.statista.com/statistics/871513/worldwide-data-created/#:~:text=The%20total%20amount%20of%20data,replicated%20reached%20a%20new%20high.. – Zugriffsdatum: 02.06.2022

[26] Sun, Mike: *RDDs vs DataFrames vs DataSets: The Three Data Structures of Spark*. 2020. – URL https://www.wisewithdata.com/2020/05/rdds-vs-dataframes-vs-datasets-the-three-data-structures-of-spark. – Zugriffsdatum: 23.03.2022

A Appendix

A.1 LOC analysis Spark versus Flink

```
❯ gocloc spark
```

Language	files	blank	comment	code
Scala	4241	134649	230227	740110
Plain Text	1057	24215	0	151181
Python	500	33003	72985	109126
Java	1058	17691	37490	85406
Markdown	210	10809	0	45014
SQL	423	5463	12812	26563
Q	1557	11463	15262	26174
JSON	74	1	0	25161
R	97	3734	14887	17055
Maven	36	329	1119	8405
ReStructuredText	53	1704	0	7365
JavaScript	34	607	1224	4672
BASH	94	1011	2129	3764
Jupyter Notebook	2	0	0	3272
YAML	62	326	724	1896
CSS	18	289	235	1303
RMarkdown	1	365	0	929
HTML	10	54	229	868
XML	13	157	290	838
Ruby	3	67	61	201
Batch	19	93	383	193
PowerShell	1	30	45	67
Bourne Shell	3	9	53	28
C	1	10	20	19
Makefile	2	10	35	17
TOML	1	2	19	13
TOTAL	9570	246091	390229	1259640

Figure A.1: Apache Spark LOC analysis conducted with gocloc (own illustration)

```
❯ gocloc flink
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                         13219         321522         466606        1497404
Scala                         1313          34718          49151         214309
Markdown                       621          38493              0         139724
XML                            263           2091           4171         113104
Python                         203          11313          17387          34499
JSON                            15              2              0          30819
Maven                          207           3415           4823          27152
HTML                           188            243           1179          17343
BASH                           130           2193           3442           8009
YAML                            24            304            697           6097
TypeScript                     122            821           2298           6094
SQL                            136            148            362           5260
CSS                              4              6             31           2281
Cython                          14            508            582           2238
LESS                            50            283            848           1352
Plain Text                      19             17              0            481
Protocol Buffers                 3             97             73            407
JavaScript                       8             50            230            346
Q                               22            600           2886            280
Bourne Shell                     1             34             62            220
Batch                            2             42              0            200
ReStructuredText                 6             50              0            195
Sass                             2             48             32            188
Makefile                         1             20             25             92
TOML                             1             21             31             57
C Header                         1              0             16              1
-------------------------------------------------------------------------------
TOTAL                        16575         417039         554932        2108152
-------------------------------------------------------------------------------
```

Figure A.2: Apache Flink LOC analysis conducted with gocloc (own illustration)

A.2 Out of memory error on Flink node hypothesis 1

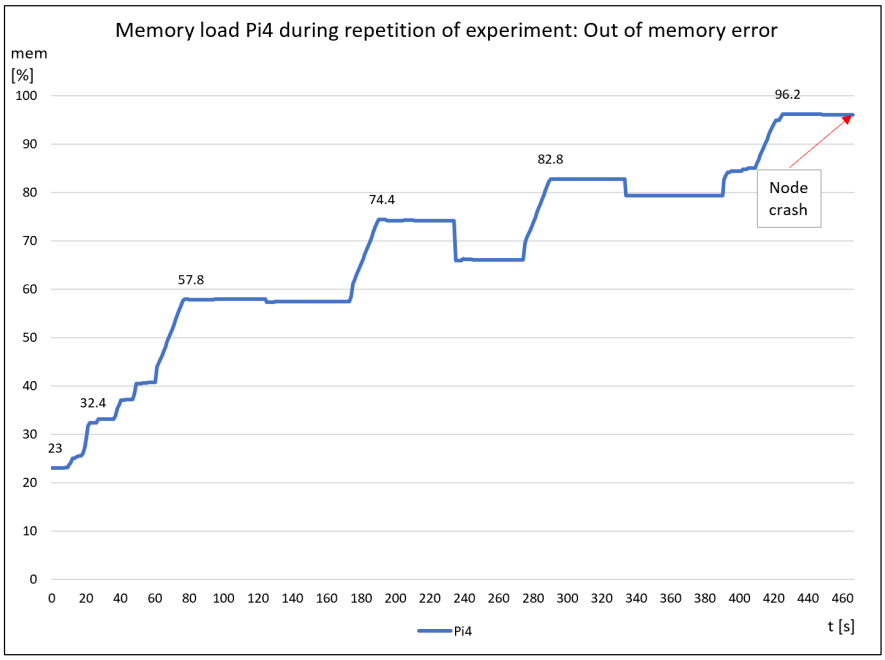

Figure A.3: Out of memory error on Pi 4 as Flink worker node upon repetition of experiment (own illustration)

A.3 Hypothesis 3: Memory and bandwidth load on Pi 4

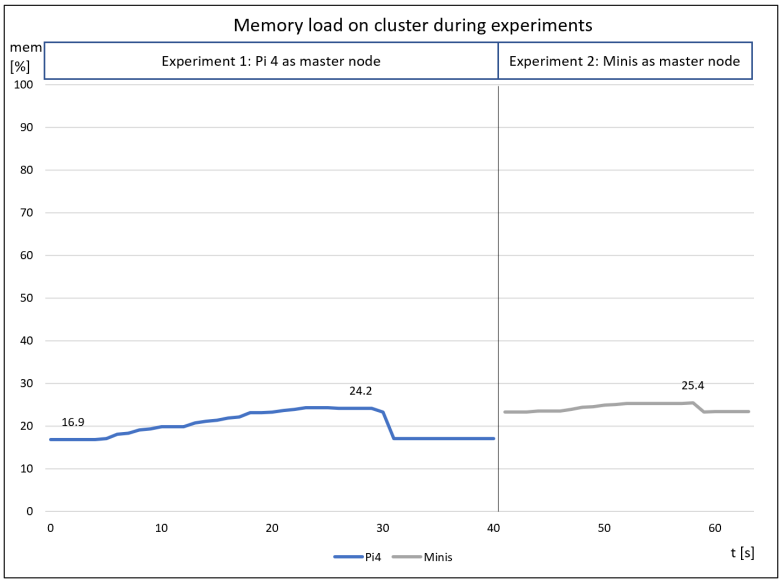

Figure A.4: Memory utilization is not causing bottleneck on Pi 4 master node (own illustration)

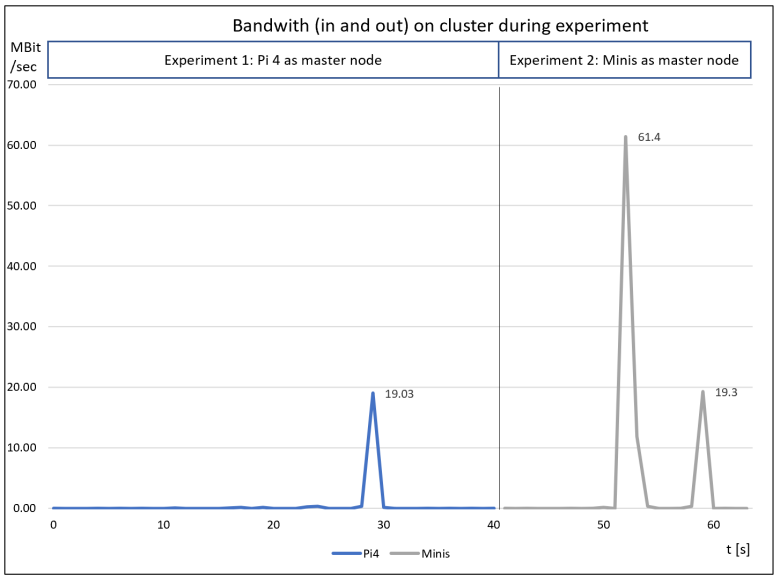

Figure A.5: Network utilization is not causing bottleneck on Pi 4 master node (own illustration)

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original