Assignment of bachelor’s thesis

Title: Application for tuning of Spark processes
Student: Ing. Jan Šťovíček
Supervisor: Ing. Marek Sušický
Study program: Informatics
Branch / specialization: Web and Software Engineering, specialization Software Engineering
Department: Department of Software Engineering
Validity: until the end of summer semester 2020/2021

Instructions

The main goal of the thesis is to design and implement an application to support performance tuning in Apache Spark. In the first part, there should be a review of existing solutions and identification of their weaknesses and strong features. With this knowledge, in the second part, design and implement a tool for the support of the tuning. The tool will use information about finished and running tasks provided by Spark History Server and will detect the most common issues like disk spills and data skews.

The application should have a web interface, where the user can see a list of jobs, he will be notified about unhealthy runs and he will be able to compare multiple runs of one specific job.

Electronically approved by Ing. Michal Valenta, Ph.D. on 9 December 2019 in Prague.
Bachelor’s thesis

Application for tuning of Spark processes

Ing. Jan Šťovíček

Department of Software Engineering
Supervisor: Ing. Marek Sušický

May 13, 2021
Acknowledgements

I would like to thank to Ing. Marek Sušický for his supervision, for all the valuable discussions and for everything he taught me about the Big Data world in the recent years. Nevertheless, I would also like to thank to my family and especially to my wife for all the endless support.
I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 13, 2021
Abstrakt

Apache Spark je v současnosti jedním z nejpopulárnějších nástrojů pro zpracování a analýzu velkých dat. Je založen na distribuovaném zpracování dat na clusterech. Právě na sdílených clusterech, ale nejen na nich, je velmi důležitý výkon aplikací.

Předkládaná práce se zabývá návrhem a vývojem open-source aplikace Sparkscope, která slouží k podpoře optimalizace a ladění výkonu jednotlivých Spark aplikací. Součástí práce je i analýza nejčastějších problémů majících vliv na výkon Spark aplikací. Sparkscope se snaží tyto problémy u konkrétních aplikací detekovat a vizualizovat. Díky tomu napomáhá optimalizovat provoz na sdílených clusterech.

Klíčová slova  Apache Spark, Big Data, ladění výkonu, optimalizace.
Abstract

Apache Spark is one of the most popular tools for big data analytics and processing. It’s based on a distributed data processing on clusters. Especially on the shared clusters, the performance of the applications is of great importance.

The presented thesis focuses on a design and development of an open-source application called Sparkscope, which can be used for a support of optimization and performance tuning of the Spark applications. An analysis of the most common performance issues of Spark applications was also done. Sparkscope aims to detect and visualize these issues for the existing Spark applications. With this approach, it helps to optimize the workload on the shared clusters.

Keywords   Apache Spark, Big Data, performance tuning, optimization.
## Contents

**Introduction**

1 **State-of-the-art**

1.1 General overview of Apache Spark

1.1.1 Spark Components ................................. 3
1.1.2 RDDs, DataFrames and Datasets ..................... 4
1.1.3 Spark cluster architecture .......................... 4
1.1.4 Spark application scheduling and execution ......... 6
  1.1.4.1 Spark application ............................. 6
  1.1.4.2 Spark job .................................... 6
  1.1.4.3 Stage ........................................ 6
  1.1.4.4 Task .......................................... 7

1.2 Known tuning tools for Spark .......................... 7

1.2.1 Spark History Server ................................. 7
1.2.2 Dr. Elephant ....................................... 8
1.2.3 Sparklens .......................................... 9
1.2.4 Sparklint .......................................... 9
1.2.5 Cloudera Workload XM .............................. 10

2 **Analysis**

2.1 Challenges for Spark applications ................. 11

2.1.1 Executor Size ..................................... 12
2.1.2 Memory Optimization .............................. 13
2.1.3 Multitenancy ....................................... 14
2.1.4 Static and Dynamic Allocation .................... 14
2.1.5 Serialization ...................................... 15
2.1.6 Joins ............................................. 15
2.1.7 Stage Skew ........................................ 16
2.1.8 Lazy evaluation .................................... 16
2.2 Requirements

2.2.1 Functional Requirements

2.2.1.1 F1 – Data Storage

2.2.1.2 F2 – Overview of Spark Applications

2.2.1.3 F3 – Filtering Applications

2.2.1.4 F4 – Detect the issues of the Spark applications

2.2.1.5 F5 – Application Details View

2.2.1.6 F6 – Compare View

2.2.1.7 F7 – Application History View

2.2.2 Non-functional Requirements

2.2.2.1 N1 – Open Source

2.2.2.2 N2 – Extensibility

2.2.2.3 N3 – Graphical User Interface

3 Design

3.1 Application Architecture

3.2 Database Design

3.3 UI Design

4 Implementation

4.1 History Fetcher

4.2 Heuristics

4.2.1 Job or Stage Failure Heuristics

4.2.2 Stage Skew Heuristic

4.2.3 Disk Spill Heuristic

4.2.4 Driver or Executor Garbage Collector Heuristics

4.2.5 Serializer Config Heuristic

4.2.6 Dynamic Allocation Config Heuristic

4.2.7 Dynamic Allocation Min/Max Executors Heuristic

4.2.8 YARN Queue Heuristic

4.2.9 Memory Configuration Heuristic

4.2.10 Core Number Heuristic

4.3 Presentation Layer

4.4 Further Development

5 Testing

Conclusion

Bibliography

A Installation Guide

B History Fcher User Guide
C Acronyms 53
D Contents of enclosed DVD 55
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Architecture of Spark on a cluster</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture of Sparkscope</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Sparkscope database model</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Sparkscope search page</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Sparkscope compare page</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Metrics class diagram</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Sparkscope search page screenshot</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Sparkscope compare page screenshot</td>
<td>38</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Spark History Server REST API endpoints</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Construction of the primary keys</td>
<td>31</td>
</tr>
</tbody>
</table>
Introduction

In the recent years, the amount of data being created in various fields of human activities is successively and rapidly growing. In 2020, the amount of data in the digital universe is estimated to be around 44 zettabytes ($10^{21}$ bytes), tenfold more than in 2013 $^1$. This is raising new challenges not only for humans themselves, but also for computational processing of the data (although the technologies for data processing, network transfer or reading from and writing to disks are evolving as well).

The term 'Big Data' has been used since 1990s. It does not refer only to high amounts of data, but also to new ways of working with these data $^2$. These new ways also comprise many tools for working with Big Data.

One of those is Apache Hadoop, which is a name of a whole ecosystem comprising various components (the basic three ones are YARN, HDFS and MapReduce). Without diving too deep in details, we can just state that Hadoop is a platform based on distributed data storage and distributed data processing, becoming a highly scalable alternative to traditional RDBMS systems.

Another Big Data tool I would like to mention is Apache Spark, which started in 2009 as a research project in the UC Berkeley AMPLab (formerly known as RAD Lab) and became a powerful alternative to MapReduce. In 2010, Spark became an open source project, and since 2013, it was transferred to the Apache Software Foundation $^3$. Since then, it has evolved into one of the most popular tools for data processing, being frequently used on Hadoop clusters.

However, despite the power of Apache Spark, the underperformance of Spark applications is a very common problem. This arises from the complicated architecture of Spark, numerous configuration items (sometimes described as 'too many knobs problem'), but also from the complexity of optimization of the distributed computing jobs.

Given those aspects, it might be difficult to tune the applications properly, or even to recognize the application is underperforming. This can lead to
Introduction

consuming more resources than necessary, and hence, generating extra costs. Therefore, a tool for monitoring of Spark jobs performance could be useful for developers in order to identify these underperforming jobs, find their bottlenecks and get a hint how to improve their performance.

Within this thesis, I aimed to design and implement such a tool. Although similar applications are already available, there are still some gaps. In the next chapter, I will summarize the basic principles of Apache Spark, analyze what are the most common issues in Spark applications, and provide brief overview of available tools for support of optimization of Spark applications. The rest of the thesis will focus on the analysis, design and implementation of the tool itself.
1.1 General overview of Apache Spark

As stated in the introduction, nowadays, Spark is one of the most popular tools for data processing. With over 1400 contributors, it has also been one of the most active projects within The Apache Software Foundation. Although it can run also locally, Apache Spark was designed as an engine for distributed data processing. Operating on Apache Mesos, Hadoop YARN or Spark Standalone cluster managers, it utilizes the individual nodes of the cluster not only for computation, but also for the data storage. Furthermore, it allows to cache the data in memory of the individual nodes, gaining a significant performance boost. Its high-level APIs for Scala, Java, Python (named PySpark) and R (SparkR) make Spark highly accessible and integrable to programs in any of these languages.

1.1.1 Spark Components

Spark comprises several integrated components, which are designed to interoperate closely, and which act like libraries in a software project:

- **Spark Core** contains the basic Spark functionality (among others, a task scheduling, memory management, fault recovery or data storage).

- **Spark SQL** is a package allowing querying data via SQL or SQL-like variants. It enables to access data from several data sources (Hive tables, JSON, Parquet, Avro or more).

- **Spark Streaming** enables processing of streams of data, making a step towards realtime data processing.

- **MLLib** is a library providing common machine learning functionality, such as classification, regression or clustering methods.
GraphX supports graph processing, allowing to create graphs, assigning properties to each vertex and edge and offering a library or common graph algorithms.

1.1.2 RDDs, DataFrames and Datasets

Before discussing the challenges the optimization possibilities for Spark applications, it’s necessary to describe how Spark stores and handles the data.

Large datasets are represented as Resilient Distributed Datasets (usually abbreviated as RDD), a main data abstraction used by Spark. An RDD is an immutable collection of objects (called partitions). As Spark usually operates on a cluster, the partitioning brings two important features:

- The partitions can be stored on different machines.
- Several partitions of a single RDD can be processed in parallel.

There are two types of operations on RDDs used by Spark:

**Transformations** are the operations returning another RDD. Transformations include e.g. `map`, `filter`, `join`, `distinct` or `reduceByKey` methods.

**Actions** are the operations which return anything else but RDD. Actions are represented by e.g. `count`, `collect`, `take` or `foreach` methods.

RDDs are evaluated lazily. This means no calculations are started until an action is called. This can bring performance benefits which will be discussed in the next sections.

Besides RDDs, newer versions of Spark uses two higher level abstractions, DataFrames and Datasets, introduced in Spark 1.3 and 1.6, respectively.

On the conceptual level, **DataFrame** is equivalent to a table in RDBMS. The data in DataFrame are organized in columns. Just like RDDs, the DataFrame API has transformations and actions. On top of the RDDs, DataFrames can benefit from in-built optimizations. DataFrame is an untyped API. Therefore, the data types can be checked only in runtime.

This flaw was fixed by **Dataset** API, which is compile-time type safe. Since Spark 2.0.0 version, these APIs (DataFrames and Datasets) are unified as Datasets. However, Python and R are dynamically typed languages (unlike Java and Scala) so in these languages, Dataset is not supported and DataFrame is used instead.

1.1.3 Spark cluster architecture

Assuming the resulted application will be used on clusters where YARN is used as a resource manager, we can omit the description of how Spark applications
1.1. General overview of Apache Spark

Spark runs in a master-slave architecture. A Spark application is driven by driver program. Most of the actual computation load is usually distributed among worker nodes, the physical nodes within the cluster. The driver, as well as the worker nodes, communicate with cluster manager. The overview of Spark architecture on a cluster is depicted on Figure 1.1.

Based on the mode used, the driver can run either inside an application master process managed by YARN (cluster mode) or on the machine from which the job was submitted (client mode). The driver is a central coordinator of the Spark application. It is the process which runs the main function and creates the SparkContext (an object representing the connection to a Spark cluster and holding the Spark configuration. Since Spark 2.0.0, it is wrapped in SparkSession object). A Spark application corresponds to one instance of SparkContext or SparkSession.

Once SparkContext is started, the cluster manager launches one or more executors on the worker nodes. An executor is a JVM (Java Virtual Machine) with allocated resources (CPU cores, memory/cache). The computational tasks are distributed among the executors. Besides the calculation itself, the executors are also used for in-memory data storage. Every Spark application has its own executors. An executor cannot span multiple nodes. On the other hand, one node can contain more than one executor.

Since version 1.2, Spark offers static and dynamic resource allocation. With the static allocation, an application allocates a fixed amount of resources and keeps it for the whole application lifetime. The dynamic allocation allows to add or remove executors during the application lifetime based on the application’s needs.
1.1.4 Spark application scheduling and execution

In this section, I would like to describe the execution hierarchy of the individual components of Spark applications, as this is a key to understanding the Spark application metadata from Spark History Server.

1.1.4.1 Spark application

As mentioned before, a Spark application corresponds to an instance of SparkContext or SparkSession. A Spark application consists of one or more Spark jobs. Each job can contain one or more stages. Each stage includes one or more tasks (usually many of them), which are the smallest units of work in Spark.

1.1.4.2 Spark job

Due to the lazy evaluation used in Spark applications, no actual calculations are launched until an RDD action is called. Once this happens, a Spark job is scheduled. In other words, one Spark job corresponds to one RDD action. When RDD action is called, Spark constructs Directed Acyclic Graph (DAG) based on the RDD dependencies. The layer responsible for constructing the DAG is called the DAG Scheduler. As transformations return another RDD and actions don’t, actions can be considered leaves within the DAG [5].

1.1.4.3 Stage

Spark jobs can be broken down into stages. In order to understand how it is split, it’s necessary to distinguish between narrow transformations and wide transformations:

Narrow transformation is a transformation in which each partition of the child RDD depends on either one or more of partitions of the parent RDD. However, these dependencies have to be determinable in the design time, regardless of the values of the records. Typically, each partition of the child RDD depends on one partition of the parent RDD (and each parent has at most one child partition), or each child partition depends on a unique subset of the parent partitions (using coalesce function).

Wide transformation is, on the other hand, a transformation which requires the data to be partitioned in a particular way. For instance, sort function needs the keys within the same range to be in the same partition. In other words, the child partition dependencies are not known in the design time, and parent partitions can have multiple child partitions.

The key difference between the narrow and wide transformation is shuffling. For the aforementioned sort function, Spark needs to shuffle the data between
the nodes, so that the necessary data are collocated. Shuffle is an expensive operation, especially for larger datasets. Therefore, a lot of performance gain can be achieved by reducing the amount and complexity of the wide transformations. To be precise, in some cases, when Spark knows that the data are already partitioned in the proper way, even wide transformations do not cause shuffles.

A stage can be considered a set of calculations which do not need any network communication (neither between executors nor between an executor and the driver). Therefore, only narrow transformation can be involved in one stage. Once a network communication is required, a new stage must be started. In the DAG, the individual stages are separated with wide transformations.

In general, more stages can run in parallel, if they have no mutual dependencies. Typically, two stages resulting in two distinct RDDs, which are about to be combined in a `join` operation, can be executed in parallel.

1.1.4.4 Task

One stage typically contains many tasks running in parallel. The task is on the lowest level of the execution hierarchy. Each task within one stage runs in one executor and runs the same code on a different part of the data. One task corresponds to one partition in the target RDD.

As the executors have configurable number of cores, multiple tasks can run in one executor in parallel. The number of executors is configurable as well, or the executors can be allocated dynamically. The maximum number of tasks which can be handled by Spark in parallel can be then determined as `number of executors × number of cores per executor`. If the number of RDD partitions, and therefore, the number of tasks, exceeds this maximum value, the extra tasks are allocated to the executors after some of the executors finish the computation of the first round of the tasks.

1.2 Known tuning tools for Spark

As the complexity of the performance tuning of the Spark applications represents a challenge to developers, several tools for support of tuning of Spark applications have been developed. I would like to briefly introduce the most popular ones, either open-source or commercial ones.

1.2.1 Spark History Server

The fundamental tuning tool is Spark History Server, which is actually a part of Spark itself.

When Spark session starts, it launches Web UI, accessible on port 4040 (or subsequent ports) of the driver node. During the application lifetime,
1. State-of-the-art

the status and resource consumption can be monitored in Web UI. After the application finishes, the metadata of the finished application are accessible on Spark History Server.

Spark History Server provides a web application with metadata of Spark applications. It provides a list of recently executed Spark applications. For each application, users can access the application metadata with various levels of granularity.

Users are able to obtain a list of used Spark configuration properties is available, as well as system properties related either to Java or OS. The web application of Spark History Server also provides metrics related to executors, jobs or stages. For stages, it also describes the statistical distribution of metrics of the individual tasks. Furthermore, it also offers a graphical representation of the execution plan of the queries used in the Spark application.

Spark History Server also provides a REST API and exposes the data related to applications, executors, jobs, stages or tasks in the JSON format. The interpretation of these data is rather challenging, as the major part of the documentation is missing. Nevertheless, the data from Spark History Server are also accessible for other applications. This API is also used by Sparkscope, the application built within this thesis.

1.2.2 Dr. Elephant

Dr. Elephant is a tool for support of monitoring and tuning of either MapReduce or Spark applications, originally created by the developers from LinkedIn, open-sourced in 2016. It aims to enable performance monitoring and tuning service, to find common mistakes and flaws, provide actionable advice and compare performance over time. It uses several heuristics in order to watch out for the most common issues. It also offers a simple and concise user interface.

It enables:

- searching and filtering processes, searching by severity
- detection of those which can be fixed easily
- monitoring the configuration and recommendation of configuration of some parameters
- alarms showing stage and job failure rates
- monitoring the distribution across executors, uses several different metrics, which helps to detect either non-optimal partitioning or bad cluster weather
- watching out for disk spills
- monitoring the health history of the job over time
1.2. Known tuning tools for Spark

Dr. Elephant is known to work with Spark 1.X versions. Since Spark 2, released in 2016, users report many compatibility issues. Although Dr. Elephant documentation does not mention any of such issues, the compatibility of Dr. Elephant with Spark 2.X is at least questionable.

In 2017, Pepperdata introduced Application Profiler, a commercial repackaging of Dr. Elephant. On top of Dr. Elephant, it offered an integration with other Pepperdata products (in order to supply more detailed answer, e.g. with cluster health context). Later on, this has evolved into Pepperdata Application Spotlight\(^9\), a complex tool for tuning processes in Spark, MapReduce, Kafka or more, running either on Hadoop or on cloud.

1.2.3 Sparklens

Sparklens by Qubole\(^{10}\) uses a different approach than Dr. Elephant. Instead of fetching and analyzing the job metadata from YARN or Spark History Server, it uses Spark Event Listener. This, however, has to be registered prior to each run of a Spark job. Sparklens is able not only to recognize the bad job performance, but in many cases, it is even able to suggest a solution:

- it’s able to predict the application wall time based on a number of used executors. Therefore, it helps the user decide whether it makes sense to increase or decrease the number of required executors, and find a balance between between high cluster utilization (with lower costs) and low wall time (with higher costs).

- it notifies the user if too much workload has been put on a driver, which reduces the parallelism, and therefore, decreases the job performance. These issues are often not very obvious. For example, Pandas, a popular data analysis library for Python, offers a structure called DataFrame, which can be converted to Spark DataFrame, and vice versa. However, one of the key differences with regards to a Spark job performance is, that Pandas DataFrame is processed by the driver, while Spark DataFrames is processed by many executors in parallel, Sparklens can detect these issues and make them obvious\(^{11}\).

Sparklens became an open-source project in 2018\(^{12}\).

1.2.4 Sparklint

Sparklint\(^{13}\) is a tool developed as a side project in Groupon, but as well as the other mentioned tools, it has become open-source. Sparklint uses the similar approach to monitoring and tuning Spark jobs as SparkLens. Using a custom Event Listener, it can analyze the individual Spark applications and monitor several metrics concerning the application performance – for instance number of executors, core utilization or idle time.
The Sparklint documentation also mentions another features being planned to be added:

- finding RDDs that can benefit from persisting
- creating automated reports on the application bottlenecks
- giving recommendations on configuration parameters, such as partition size or job submission parallelism

However, the last contribution in the Sparklint repository was committed in the beginning of 2018. Since then, no new functionalities were released (including the three aforementioned ones)\footnote{[14]}, which makes a future of Sparklint quite questionable.

1.2.5 Cloudera Workload XM

Cloudera Workload XM (Experience Manager) is a tool for monitoring of the performance of a Hadoop Cluster. It uses data from Spark History Server to diagnose Spark applications. Besides that, it can also monitor the performance of MapReduce jobs (via YARN Job History Server), Oozie workflows (via Oozie Server), Hive queries (via HiveServer) and Impala queries (via ImpalaD daemon).

As the previous paragraph suggests, Workload XM does not specialize on Spark application tuning, but monitors the overall performance of the whole cluster. Nevertheless, concerning Spark applications, it’s able to set baselines of application duration and resource usage and detect anomalies, as well as detect the most common Spark app health issues like data skews or wasting resources.

Unlike the previously mentioned tools, Cloudera Workload XM is a commercial tool, and therefore not for free.
In this chapter, I would like to describe the background of the tuning of Spark applications, which I took into account while designing the new tool, eventually named Sparkscope. Moreover, I will define the requirements on this tool.

Some of the technical details may vary with a version of Spark and with a platform it runs on. As Sparkscope is intended to be used on an on-premise platform where Spark 2.4 and Hadoop 3 are installed and YARN is used as a resource manager, all the details will be applicable for these versions. Furthermore, on this platform, Spark is mostly used for building ETL data pipelines in Spark SQL. Therefore, the following text will focus on that and may omit some specifics of Spark Streaming, MLLib or GraphX.

2.1 Challenges for Spark applications

Distributed computing is a very complex topic. Even though Hadoop or Spark provide developer-friendly ways of building applications and data pipelines, the internals remain very complicated. Very often, large data amounts are processed, forcing the applications to fulfill high requirements concerning performance and optimization. There are lots of configuration options, making the whole performance tuning process very demanding. Furthermore, the clusters are often used by several tenants who need to share the resources. This also has an impact on performance tuning.

In this chapter, I would like to summarize the points which are often subject of the performance tuning of Spark applications, or which represent the challenges and usual pain points Spark developers often need to deal with. Besides that, I would also like to describe some symptoms how the potential issues can be recognized, e.g. using Spark History Server.
2. Analysis

2.1.1 Executor Size

One of the key problems is configuring a proper size of the executor. Typically, each node of the cluster consists of several CPU cores. With the executor size, we usually mean the number of cores per one executor, configured by the `spark.executor.cores` property. Besides the CPU cores, the amount of memory allocated per one executor also can be configured. The aspects of the memory allocation will be further discussed in subsection 2.1.2, but let’s assume the amount of memory per executor will be configured proportionally to the number of cores.

An important limitation is that an executor may use the resources only from one node. Therefore, the executor cannot allocate more cores than the number of cores available on one node. Nevertheless, even with this assumption, the total number of cores per node still needs to be taken into account. For example, on a cluster with 10-core nodes, allocating 6 cores per executor means that there will be only 4 cores left on the involved nodes. These cores cannot be utilized by the same application anymore. Therefore, it makes sense to allocate the CPU cores in a way, so that the all the CPU resources are fully utilizable.

The optimization of number of cores per executor seems to be not an exact science. In other words, no precise algorithm of determination of the optimal core number is known. The developers need to rely on experiments or can utilize some empirical knowledge from the community. Nevertheless, there are some general trends which should be taken into account.

Using very small executors with very few cores (one core in extreme case) brings a benefit of being able to use many of them. On the other hand, this approach also brings disadvantages. With many small executors, there is less memory left per executor, which might lead to out-of-memory issues or spilling to disk. Furthermore, the benefit of a parallelism within one executor gets lost, as only a few tasks might be computed in one executor at a time.

Using very large executors with many cores (in extreme case, one executor per node) enables using parallelism within one executor, but too large executors tend to waste the resources, as they often cannot utilize all the cores. The official Spark documentation recommends using 2–3 tasks per one CPU core. Furthermore, the HDFS client was reported to have issues with running too many concurrent threads. With regards to that, Cloudera estimated that no more than 5 cores per executor should be used.

To avoid the abovementioned downsides of using too small or too large executors, some middle way should be chosen. Holden Karau and Rachel Warren recommend using 5 cores per executor base on their experience, but at the same time, they report having performant applications with 6 or 7 cores per executor. Moreover, the overall complexity of the application should be also taken into account. For instance, for a simple application with less input data, less cores per executor can also work well – less cores might be sufficient.
2.1. Challenges for Spark applications

for simple applications and more resources will be left for other applications.

The number of cores per driver can be also configured with property
\texttt{spark.driver.cores}. Usually, most of the calculations are run on the exec-
utor. Therefore, using larger driver rarely speeds the computation up. On
the other hand, on YARN, driver can utilize multiple cores if a part of the
the calculation is run on the driver, and since there is only one driver per appli-
cation, increasing the driver size should not take too much resources from the
executors.

2.1.2 Memory Optimization

The optimization of memory allocation is tightly connected with the executor
size mentioned above. As well as the CPU cores, the memory can be also allo-
cated via configuration. For executors, there are two important configuration
properties: \texttt{spark.executor.memory} sets memory allocated per each execu-
tor and \texttt{spark.executor.memoryOverhead} defines an amount of additional
memory per executor, which is used for virtual machine overhead, interned
strings or other overhead [6].

By default, Spark uses 10\% of the executor memory as the executor mem-
ory overhead, but 384 MB at least. The similar values are also recommended
by experienced Spark users. Larger memory overhead rarely has an effect on
the application performance [17].

Similar considerations to those mentioned in subsection 2.1.1 apply to
memory as well. Moreover, the CPU cores and memory should be spent
relatively evenly: for example, for nodes with 20 cores and 100 GB of memory,
a reasonable configuration could be \texttt{spark.executor.cores} = 5 (i.e. 25\% of
the node) and \texttt{spark.executor.memory} = 23G (slightly less than 25\% of
the node, so there is a space for the memory overhead). Uneven distribution
of CPU cores and memory could mean that either some CPU cores or some
memory will be left unusable.

Just like for the CPU cores, too little or too much memory allocated for
executor have downsides and developers should avoid it.

Obviously, allocating too little memory can lead to out-of-memory issues
and the application may even crash. Furthermore, one should keep in mind
that the minimum executor memory overhead is 384 MB. If too many small
executors are used, with \texttt{spark.executor memory} = 1 GB, about 25\% of the
memory will be used for overhead, instead of the actual computation.

On the other hand, allocating too much memory also brings its downsides.
With the amount of heap memory, a garbage collection time may arise, so the
application may suffer from garbage collection delays if too much executor
memory is allocated. The garbage collection time can be indicated in Spark
History Server.

In some cases, Spark is able to evict some blocks of data from memory to
disk. Of course, this has a negative impact on performance, but it can prevent
the application from crash. These disk spills can be also monitored in Spark History Server, and therefore, can give a hint about how the executor memory can be optimized. In case of disk spill, increasing the executor memory could help the application performance.

The driver memory configuration is usually not as crucial as the executor memory. Nevertheless, just as the executor memory, it can be configured with `spark.driver.memory` and `spark.driver.memoryOverhead` parameters. It’s possible that the driver needs to perform some large calculation or a larger amount of data needs to be collected to the driver. In such cases, the application might fail if there is not enough driver memory. Furthermore, the maximum amount of data that can be returned to the driver is defined by `spark.driver.maxResultSize` parameter. The purpose of this parameter is forcing the application to fail earlier, rather than wait for the driver out of memory error. A general recommendation for driver memory is to keep it as low as the application allows, to save more memory for executors.

2.1.3 Multitenancy

The topic of multitenancy might seem to be out of scope of optimization of Spark applications. Nevertheless, the Spark applications are often operated on shared clusters. Therefore, the resources are shared by multiple users or even multiple teams. A proper configuration of resource pools in YARN might help with a fair distribution of the resources among the tenants.

If the YARN resource pools are configured, the Spark applications can be submitted to the respective queue using the `spark.yarn.queue` parameter.

2.1.4 Static and Dynamic Allocation

The topic of static or dynamic allocation of the resources is quite closely connected to the previous subsections. In subsection 2.1.1 and subsection 2.1.2, scaling of the executors was described. The number of the executors can be determined either statically or dynamically.

For static allocation, `spark.executor.instances` parameter plays an important role, as it defines the number of executors to be allocated.

However, especially for shared or busy clusters, better overall performance might be achieved with dynamic allocation. This enables to request or decommission executors based on an application’s needs. When running into an expensive part of a computation, the application can request more resources. When these resources are no longer needed, some executors can be decommissioned and they can be utilized by other applications.

The dynamic allocation of the resources can be switched on by configuring `spark.dynamicAllocation.enabled` to `true`. Besides that, the properties `spark.dynamicAllocation.shuffleTracking.enabled` should be set to `true`. Alternatively, `spark.shuffle.service.enabled` should be set to `true`.

14
2.1. Challenges for Spark applications

and an external shuffle service should be configured, so that the dynamic allo-

cation works properly. Otherwise, issues can occur: the application might
not be able to preserve the shuffle files or it can even hang in a pending state.

2.1.5 Serialization

One of the important aspects which affect the performance of Spark applica-
tions is serialization of objects. The Spark objects need to be serialized either
when shuffling data between nodes or when the data are written to disk (for
storage or as a disk spill).

By default, Spark uses Java serialization, which is often slow. As an alter-
native, Kryo serializer [18] can be used. This is known to perform better for
RDDS [15].

Kryo serializer can be activated by setting spark.serializer to the proper
Java class name of Kryoserializer.

2.1.6 Joins

In the distributed systems, joining two tables is a very expensive operation,
as the basic algorithms require shuffling.

**Shuffled Hash Join** uses a map-reduce principle. In the first phase, the
joined datasets are shuffled, so that the records with the same keys are
located in the same partition. After that, the smaller dataset is hashed
into buckets. The keys from the larger dataset are then attempted to
match with the keys in the appropriate bucket.

**Sort Merge Join** also uses a shuffle as the first step, so that the rows with
the same keys end up in the same partition. After that, the records
within each partition are sorted by the key. Then, Spark iterates through
both sets of keys and joins the data.

**Broadcast Join**, on the other hand, does not shuffle the data. The smaller
dataset is broadcasted to the executors. The algorithm then continues
with the "hash" phase known from Shuffled Hash Join: the keys from
the smaller dataset are hashed into buckets and the records from the
larger dataset are matched to the appropriate key.

Spark itself is able to choose an appropriate algorithm based on the statis-
tics of the datasets. Therefore, it’s probably not necessary to take a deep dive
into details. Nevertheless, one point is worth mentioning. Broadcast join is the
only algorithm that does not use shuffle. Therefore, it can be very efficient, if
one of the datasets is small enough. By default, the threshold is 10 megabytes
and can be controlled with spark.sql.autoBroadcastJoinThreshold prop-
erty. Besides that, users can also force Spark to explicitly use this algorithm.
2. Analysis

2.1.7 Stage Skew

Usually, the processed datasets are split to several partitions. As described in subsection 1.1.4, processing one partition is handled within a task. Usually, multiple tasks are processed in parallel. In an ideal case, the partitions are equally sized. However, in reality, the distribution of the partitioning keys is often not equal. This is called stage skew and it is one of the most common performance issues in Spark applications.

In general, we speak about stage skew if some of the tasks within one stage are significantly larger than others. The smaller tasks can be processed very quickly and the whole stage (and all the resources) is waiting for the stragglers to finish. Therefore, stage skew prolongs the job duration, and on top of that, it can cause the executors to be idle. Stage skew can get very significant especially in joins, if the joining keys have skewed distribution.

The stage skews can be easily detected in Spark History Server, or in its API, as both provide statistical distribution of various metrics (like task duration, garbage collection time or shuffle size) over tasks within a stage. By default, for each stage and each metric, it provides five quantiles: minimum, first quartile, median, third quartile and maximum. Therefore, it’s easy to detect that one task is significantly larger than others.

Once the developer is able to predict the stage skew, based on the knowledge of the data, he can also avoid it. As we usually speak about stage skew in a context of joins, one of the easiest options is using broadcast join algorithm, if the skewed table is small enough.

Another option is salting the key. For instance, let’s consider a table having large one partition, as one key is much more abundant than the others. Appending a random number to this key can force Spark to distribute this huge partition into several smaller ones, which can be processed on multiple nodes. This can greatly increase the performance. Nevertheless, this always needs a detailed knowledge of the data being processed.

2.1.8 Lazy evaluation

Another challenge is connected to the lazy evaluation principle. As described in the previous chapters, Spark does not trigger any calculations until an action is called. Any transformation is just added to the execution plan which waits to be triggered by the action.

In some cases, caching or persisting the dataset with cache() or persist() methods might increase the application performance. This is usually valid if the same dataset is used multiple times during the calculation. With cache() or persist() used at a right place, the application can avoid running the same calculations multiple times, as it can use the a dataset stored in a memory instead of calculating a new one from scratch.
Nevertheless, caching the data does not always help. Putting the data into memory is related to some overhead. Furthermore, column-oriented storage formats like Parquet are optimized for regular SQL queries and these are fast on its own. Therefore, caching the datasets on top of Parquet files is not always helpful. Finally, large datasets might not fit into memory. If the datasets are too large, they might need additional disk I/O operation that can decrease the performance \[19\].

### 2.2 Requirements

In this section, I would like to summarize the requirements on Sparkscope. I will split them into two groups – functional and non-functional requirements.

#### 2.2.1 Functional Requirements

##### 2.2.1.1 F1 – Data Storage

The application should contain a database storage. It should be able to store the data collected from the Spark History Server API in this database and provide these data to a web application.

##### 2.2.1.2 F2 – Overview of Spark Applications

The application should provide a concise overview of the finished Spark applications on one page. For each application, several details should be visible on the overview page:

- application id
- application name
- Spark user
- start and end time of the application
- application duration
- list of the detected issues and graphical representation of their severity

##### 2.2.1.3 F3 – Filtering Applications

On the overview page, it should be possible to filter the applications based on their application id, application name, Spark user, start time and end time.
2. Analysis

2.2.1.4 F4 – Detect the issues of the Spark applications

The application should be able to detect the most common issues of the Spark applications. The detection should be based on heuristics. The application should distinguish between two levels of the severity of the issues. The heuristics should be configurable - the thresholds for either low or high severity should be possible to define in a configuration file without a necessity to change the source code.

2.2.1.5 F5 – Application Details View

For every finished Spark application, the application should provide a page with details. This page should include all the items mentioned in the Overview requirement. Besides those, it should display basic metrics like CPU usage, garbage collection time, number of jobs, stages and tasks. Furthermore, it should provide a list of the configuration properties of the particular Spark application. For each detected issue, it should display a further description (if applicable) which could help the user to solve the issue.

2.2.1.6 F6 – Compare View

The application should provide a page where user can select two application that should be displayed side by side in one screen, so that their metrics, issues and configuration can be easily comparable. For each of the two application, the page should contain the details described in the Application Details View requirement.

2.2.1.7 F7 – Application History View

The history of each recurring application (i.e. multiple instances of the application with the same name, like daily ETLs) should be also accessible. It should be displayed in a graph describing how the application duration evolves over time.

2.2.2 Non-functional Requirements

2.2.2.1 N1 – Open Source

The application should be released with an open source code.

2.2.2.2 N2 – Extensibility

The application should be easily extensible. There should be a possibility to create additional heuristics.
2.2.3 N3 – Graphical User Interface

The application should provide a GUI (Graphical User Interface) accessible via common web browsers (Mozilla Firefox, Google Chrome and Microsoft Edge).
Chapter 3

Design

Sparkscope was designed with regards to findings and requirements defined in the previous chapters. In this chapter, I will discuss the details of the application design.

3.1 Application Architecture

Sparkscope consists of two main parts.

The first part is a component called History Fetcher. History Fetcher execution can be either scheduled in a convenient tool like Cron or started on demand from command line. It collects the data from Spark History Server REST API and passes them to the Data Access Layer implemented using ORM (Object Relational Mapping). Then, the data is stored in a relational SQL database.

The second part is Web Application. This part uses the same Data Access Layer to retrieve the data from the same SQL database. The data are then provided to the Business Layer. Here, the raw data are processed in order to provide useful insights about the performance of the analyzed Spark applications. After that, the data are handed over to the presentation layer which is accessible via web browser.

The complete overview of the architecture of Sparkscope is visualized on Figure 3.1.

3.2 Database Design

The design of the database model was driven by the hierarchy of execution of Spark applications (described in subsection 1.1.4) and also by the structure of Spark History Server REST API endpoints.

The entity called Application represents an instance of a Spark application. Its attributes carry top-level information about the Spark application.
3. Design

Figure 3.1: Architecture of Sparkscope. The Spark logo from the official Apache Spark website [6].

– its duration, start and end time, application id and name, the user who executed it, but also a complete list of configuration properties.

The Executor entity has N:1 relationship with the Application entity, as the Spark application can usually spin up multiple executors. This entity contains lots of attributes related to memory usage. Nevertheless, the memory usage metrics turned out to be not accessible on Spark History Server API (this will be described in the following chapters). Besides that, it provides attribute carrying information about the duration, garbage collection time, amount of input data, shuffle data, number of tasks done by the executor or number of its cores.

As the Spark application usually includes multiple jobs, the Application entity also has a 1:N relationship to Job entity. This possesses several attributes related to number of stages and tasks (total, finished, skipped, failed or killed), submission and completion time, the job status or others.

The Stage entity has N:1 relationship with Application and Job entities and it contains detailed information about each stage – several attributes
3.2. Database Design

Figure 3.2: Sparkscope simplified database model with selected attributes. The bold attributes are the primary keys. The joining keys are highlighted with light blue. The schema was created with dbdiagram.io.

related to duration (executor run time, CPU time, submission and completion time), amount of input or output data (records and bytes), amount of shuffle data, amount of data spilled to disk, failure reason if applicable and more.

Stage_Statistics entity has 1:1 relationship with Stage entity. The Stage_Statistics entity represents a statistical distribution of task metrics within a stage. For each attribute, it provides an array of five values. These values represent five quantiles: 0.001 (basically, the minimum), 0.25, 0.5 (median), 0.75 and 0.999 (basically, the maximum). As each stage may consist of many task, it might be handy to get information about statistical distribution of these task metrics in a stage without a need for accessing enormously detailed data about tasks. It also provides useful information about stage skew. The attributes contain information about time metrics (including detailed breakdown to total runtime, CPU time, time for serialization and deserialization, garbage collection or getting results), detailed statistics about shuffle or about amount of input and output data.

Task entity describes tasks, the smallest units of work in Spark applications. Task entity has N:1 relationship with Stage and Executor, as one stage usually consists of many tasks and one executor can work on multiple tasks. This entity has very similar attributes as Stage_Statistics entity,
3. Design

but instead of the arrays describing the statistical distribution, the attribute contain single values for each task.

Finally, the Stage Executor entity serves as a relational entity between Stage and Executor entities and contains stage details that are specific to a concrete executor.

The database schema can be found on Figure 3.2. As the complete schema contains many attributes, the depicted schema was simplified and contains only the keys.

3.3 UI Design

The user interface was designed to have five distinct pages. The wireframes were designed in Figma [21]. On the top of each page, there is a navigation bar which should facilitate the orientation on the website. The control elements like forms were always placed on the left side.

The title page, or Dashboard, contains several statistics of how many Spark applications, executors, jobs, stages or tasks were analyzed, as well as the timestamp of the newest application. Finally, it contains three buttons that which can user use to be redirected to the Search, Compare or App History pages.

The Search page (see the wireframe on Figure 3.3) serves as a high-level overview of the analyzed applications, but it also contains a menu which enables filtering of these applications. The user can filter the application by its name, its ID, its user, and can specify the interval when the application should have started or ended. The intervals can be also set with quick-filter buttons to predefined values. In the overview, up to 50 applications can be displayed based on the filters used. Each application is represented with a panel with several basic attributes: the application ID (working as a hyperlink), its name, the user, start and end time and the duration. Furthermore, the detected issues are displayed. If an issue is detected, a banner with the issue description appears. The severity of the individual issues is represented with the banner color: yellow for low severity, red for high severity. More details can be displayed when the mouse cursor hovers over the banner. The algorithms for the detection of the issues will be elaborated in Chapter 4. The left border of each application panel is colored in a traffic light fashion: green for application with no issues detected, yellow for low severity issues only and red for applications with high severity issues.

Clicking the application ID link will open the Application page. This page contains details about a specific Spark application. Besides the information contained on the previous page, it also contains buttons for redirection to the Application History page and also to the corresponding application page on Spark History Server. Furthermore, the page provides more details split to several sections. In the Basic Metrics section, several metrics can be found.
3.3. UI Design

Figure 3.3: Sparkscope search page.
3. Design

for quick indication of the application complexity. The Issues section contains the issue banners known from the Search page, but this time, these are supplemented with further details of the issues. The Key Configurations Items section consists of some of the most critical Spark configuration properties and their values for the specific Spark application. These are mostly related to memory and CPU allocation. The last section is called All Configuration Items and it contains the complete list of the configuration properties. As this can get very long, a toggle button for collapse or expand was added.

On the Compare page, user can fill ID’s of two distinct Spark applications and can get a comparison of the performance of these two applications. A similar layout as for the Application page is used, both applications details are displayed side by side and the respective sections are aligned side by side to allow easy comparison. See the wireframe on Figure 3.4.

Finally, Sparkscope provides a App History page. The user can land here either via the navigation bar or via the button on the Application page. In the first case, the user needs to enter an application name into a left side form to see a history of the application. In the latter case, the user is redirected straight to the history page of the specific application. On this page, the user can find a plot describing how the application performance (or duration, to be more specific) evolves over time. This can be particularly useful for applications running periodically (e.g. daily ETL jobs). The plot contains all the applications having the same application name. Hovering over a specific point can display details about the specific instance of the application. Clicking the point will redirect the user to the application page.
Figure 3.4: Sparkscope compare page.

<table>
<thead>
<tr>
<th>Sparkscope</th>
<th>Dashboard</th>
<th>Search</th>
<th>Compare</th>
<th>App History</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Sparkscope, Dashboard, Search, Compare, App History**

Application id #1

Application id #2

---

**Application, 123456/7890123, 1234 (my, lovely, spark, application) [Link to SHS]**

**Start:** 2020-12-11 15:49:22 GMT

**End:** 2020-12-11 15:27:39 GMT

**Basic metrics**

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>21 min 11 s</td>
</tr>
<tr>
<td>CPU time</td>
<td>7 hr 35 min</td>
</tr>
<tr>
<td>Total tasks time</td>
<td>4 hr 42 min</td>
</tr>
<tr>
<td>Executor peak memory</td>
<td>18 GB / 20 GB</td>
</tr>
<tr>
<td>Jobs number</td>
<td>1</td>
</tr>
<tr>
<td>Stages number</td>
<td>19</td>
</tr>
<tr>
<td>Tasks number</td>
<td>2264</td>
</tr>
<tr>
<td>Total GC time</td>
<td>21 min 11 s</td>
</tr>
</tbody>
</table>

**Issues**

- **Disk Spill:** 1.6 GB spilled to disk in Stage 21
- **Data Skew:**
  - Stage 33 suffers from skew (min: 0.05 s, med: 0.32 s, max: 5.7 s)
  - Stage 35 suffers from skew (min: 0.07 s, med: 0.44 s, max: 12.1 s)

**Key configuration items**

<table>
<thead>
<tr>
<th>Memory</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver memory:</td>
<td>3 GB</td>
</tr>
<tr>
<td>Driver memory overhead:</td>
<td>1 GB</td>
</tr>
<tr>
<td>Executor memory:</td>
<td>1 GB</td>
</tr>
<tr>
<td>Executor memory overhead:</td>
<td>1 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of executors:</td>
<td>4</td>
</tr>
<tr>
<td>Cpu per executor:</td>
<td>4</td>
</tr>
<tr>
<td>Dynamic allocation:</td>
<td>true</td>
</tr>
<tr>
<td>Driver max result size:</td>
<td>4 GB</td>
</tr>
</tbody>
</table>

**All configuration items**

- **Configuration A**
  - spark.sqlrelation: org.apache.spark.sqlrelation.mkTable
  - spark.sql.resolution: org.apache.spark.sql.resolution.mkTable
  - (etc., this list might be a bit long, hence the expand/collapse button)

---

**Application, 123456/7890123, 1234 (my, lovely, spark, application) [Link to SHS]**

**Start:** 2020-12-11 15:49:22 GMT

**End:** 2020-12-11 15:27:39 GMT

**Basic metrics**

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>21 min 11 s</td>
</tr>
<tr>
<td>CPU time</td>
<td>7 hr 35 min</td>
</tr>
<tr>
<td>Total tasks time</td>
<td>4 hr 42 min</td>
</tr>
<tr>
<td>Executor peak memory</td>
<td>18 GB / 20 GB</td>
</tr>
<tr>
<td>Jobs number</td>
<td>9</td>
</tr>
<tr>
<td>Stages number</td>
<td>19</td>
</tr>
<tr>
<td>Tasks number</td>
<td>2264</td>
</tr>
<tr>
<td>Total GC time</td>
<td>31 min 11 sec</td>
</tr>
</tbody>
</table>

**Issues**

- **Disk Spill:** 1.6 GB spilled to disk in Stage 21
- **Data Skew:**
  - Stage 33 suffers from skew (min: 0.05 s, med: 0.32 s, max: 5.7 s)
  - Stage 35 suffers from skew (min: 0.07 s, med: 0.44 s, max: 12.1 s)

**Key configuration items**

<table>
<thead>
<tr>
<th>Memory</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver memory:</td>
<td>3 GB</td>
</tr>
<tr>
<td>Driver memory overhead:</td>
<td>1 GB</td>
</tr>
<tr>
<td>Executor memory:</td>
<td>1 GB</td>
</tr>
<tr>
<td>Executor memory overhead:</td>
<td>1 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of executors:</td>
<td>5</td>
</tr>
<tr>
<td>Cpu per executor:</td>
<td>4</td>
</tr>
<tr>
<td>Dynamic allocation:</td>
<td>true</td>
</tr>
<tr>
<td>Driver max result size:</td>
<td>4 GB</td>
</tr>
</tbody>
</table>

**All configuration items**

- **Configuration A**
  - spark.sqlrelation: org.apache.spark.sqlrelation.mkTable
  - spark.sql.resolution: org.apache.spark.sql.resolution.mkTable
  - (etc., this list might be a bit long, hence the expand/collapse button)
Chapter 4

Implementation

The following chapter should focus on how the distinct parts of Sparkscope were implemented. It should describe which technologies were used and which challenges did I need to cope with during the development. Furthermore, I would like to describe the used heuristics in more detail. Finally, several ideas for a further development will be introduced.

4.1 History Fetcher

History Fetcher is a component which is responsible for fetching the data from Spark History Server to a database. Spark History Server exposes a REST API where it provides Spark application data. As the data are accessible only for limited time, it makes sense to store them in a database.

The History Fetcher code was written in Python 3.6\[22\]. PostgreSQL\[23\], a classical relational SQL database system was chosen as a database storage. The data access layer was realized with SQLAlchemy\[24\], a popular object-relational mapping toolkit for Python. The History Fetcher does not provide any graphical interface. It can be started and controlled from command line.

Spark History Server API provides several endpoints with data in JSON format\[6\]. History Fetcher fetches data from some of them via GET requests. See the list of the used endpoints in Table 4.1.

The API provides a list of applications which can be filtered based on various criteria. One of them is the end date. This enables fetching only those applications which are not stored in the Sparkscope database yet, and allows incremental updates, either scheduled or ad hoc. There is also an option of a test mode which enables fetching a fixed number of application, and an option of truncating the database before the execution of the History Fetcher.

As the Table 4.1 suggests, some endpoints need to be called for every single application or even for every single stage. This implies that many HTTP requests need to be sent to the server. Doing this sequentially might
4. Implementation

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Data content</th>
</tr>
</thead>
<tbody>
<tr>
<td>/applications?status=completed &amp;minEndTime=&lt;timestamp&gt;</td>
<td>A list of the finished applications ending after the given date/time</td>
</tr>
<tr>
<td>/applications?status=completed &amp;limit=&lt;number&gt;</td>
<td>A list of the finished applications limited to the specified number of the newest applications (for initial load and testing purposes)</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/environment</td>
<td>Environment and config for the specified client mode application</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/environment/&lt;attempt_id&gt;</td>
<td>Environment and config for the specified cluster mode application</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/allexecutors</td>
<td>A list of all executors for the given application</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/jobs</td>
<td>A list of all jobs for the given application</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/stages</td>
<td>A list of all stages for the given application</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/stages/&lt;stage_id&gt;</td>
<td>Details for the given stage attempt, including a breakdown to executors</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/stages/&lt;stage_id&gt;/&lt;attempt_id&gt;/taskSummary</td>
<td>Distribution of the stage metrics over tasks, with the given quantiles</td>
</tr>
<tr>
<td>/&lt;app_id&gt;/stages/&lt;stage_id&gt;/&lt;attempt_id&gt;/taskList</td>
<td>List of the tasks for the given stage attempt. The tasks are sorted in descending order and their number can be limited by configuration.</td>
</tr>
</tbody>
</table>

Table 4.1: Spark History Server REST API endpoints used by History Fetcher. The URLs of the endpoints are relative to the base URL http://<server-url>:<port>/api/v1

be very time-consuming. As this is an I/O-bound operation, asynchronous execution using concurrent.futures library can help with it. Instead of sending a request and waiting for a response, Python can spawn multiple threads. A thread sends a request, and as it waits for response, another thread can send another request. This is not a real multithreading known from other languages like C++ or Java (threads do not work in parallel), but for I/O-bound operations, it can be very effective. The size of the threads in the pool is configurable.

Spark History Server provides an ID for each instance of each entity (like application, job, stage, task or executor). These ID’s are not entirely unique. For example, the stage ID is unique within the application, but not within
4.2. Heuristics

Table 4.2: Construction of the primary keys

<table>
<thead>
<tr>
<th>Entity</th>
<th>Primary key</th>
</tr>
</thead>
<tbody>
<tr>
<td>application</td>
<td>&lt;app_id&gt;</td>
</tr>
<tr>
<td>executor</td>
<td>&lt;app_id&gt;_&lt;executor_id&gt;</td>
</tr>
<tr>
<td>job</td>
<td>&lt;app_id&gt;_&lt;job_id&gt;</td>
</tr>
<tr>
<td>stage</td>
<td>&lt;app_id&gt;_&lt;stage_id&gt;</td>
</tr>
<tr>
<td>stage_statistics</td>
<td>&lt;app_id&gt;_&lt;stage_id&gt;</td>
</tr>
<tr>
<td>stage_executor</td>
<td>&lt;app_id&gt;<em>&lt;stage_id&gt;</em>&lt;executor_id&gt;</td>
</tr>
<tr>
<td>task</td>
<td>&lt;app_id&gt;<em>&lt;stage_id&gt;</em>&lt;task_id&gt;</td>
</tr>
</tbody>
</table>

the entire server. The task ID is unique within the stage, but not within the application. Therefore, I established a system of primary keys derived from the ID’s based on concatenation. The derivation is described in Table 4.2.

4.2 Heuristics

As discussed earlier, the performance of Spark applications is affected by many factors, including the hardware, the code, the configuration or the data itself. As the interplay between all the factors can be very complex, deriving a single universal algorithm for diagnostics and optimization of the Spark applications could be extremely challenging or even impossible. Therefore, several heuristics were developed, in order to help with identification of the potential issues. These heuristics work as the business layer.

The heuristics use four classes (AppConfigAnalyzer, ExecutorAnalyzer, StageAnalyzer and JobAnalyzer), which all inherit from the generic Analyzer class. These classes use several methods to calculate the specific metrics and return an instance of the child class of the parent AbstractMetric class: either an instance of EmptyMetric class if no issue is detected, or an instance of some concrete metric (e.g. StageSkewMetric or StageDiskSpillMetric) if an issue is detected (see Figure 4.1).

Most of the heuristics also provide two distinct levels based on severity. If an issue is detected, then the Metric object can have either low severity (depicted in yellow in GUI) or high severity (depicted in red). The thresholds for low or high severity can be managed in the configuration files.

The calculations of the heuristics is triggered every time the Application entity is accessed. This is realized using @orm.reconstructor decorator from SQLAlchemy library. It enables to change the configurations without a need to restart the server. It could also allow to have a separate configuration for each user (this feature is not implemented yet, but it’s one of the possible improvements). Unfortunately, it can have negative impact on performance, as it generates a lot of traffic between the business layer and the database.
4. Implementation

![Metrics class diagram](image)

Figure 4.1: Metrics class diagram. The `Metric` class has 12 children in total, but only two of them were depicted in the schema for the sake of simplicity and clarity.

Just like the History Fetcher, the business layer is written in Python 3.6 and has a dependency on the data access layer.

4.2.1 Job or Stage Failure Heuristics

If any of the jobs within a Spark application fails or is killed, a `JobFailureMetric` object with high severity is created and the user gets a message that some number of the jobs failed.

For stages, the situation is very similar. If a stage fails or is killed, a `StageFailureMetric` object with high severity is created. Besides the message about the failed or killed stages, the user can also see a log snippet indication the possible reason of the stage failure.

4.2.2 Stage Skew Heuristic

Stage skews were described in subsection 2.1.7. The `StageStatistics` entity provides an easy and convenient way how to detect stage skews.

First, Sparkscope is able to filter only long-running stages. If the stage duration is only a few milliseconds, a stage skew usually does not do any harm. Therefore, it may be not worth analyzing this stage. The user can set up the threshold for the stage duration in a config file.

The `StageStatistics` entity provides several attributes related to the stage performance. Among others, `executor_run_time`, `bytes_read`, `bytes_written`, `shuffle_bytes_read` and `shuffle_bytes_written`. Each
of the attributes is an array containing five quantiles of the particular task statistics within the stage. This heuristic uses maximum and median. It calculates the \( \frac{\text{max}(\text{executor\_run\_time})}{\text{median}(\text{executor\_run\_time})} \) ratio for each filtered stage. This is compared with the configurable thresholds. If at least one of the stages reaches the threshold for low or high severity, a \texttt{StageSkewMetric} object indicating the severity is created. Furthermore, the information about the distribution of the remaining four attributes are also displayed for each relevant stage. The stages with skews are sorted by the \( \frac{\text{max}(\text{executor\_run\_time}) - \text{median}(\text{executor\_run\_time})}{\text{median}(\text{executor\_run\_time})} \) difference in descending order, keeping the stage with highest potential for improvement at the top.

### 4.2.3 Disk Spill Heuristic

If Spark does not have enough memory for storing objects, it can spill a part of the memory to disk. This can lead to performance issues and Sparkscope is able to detect that.

Here, it might be worth it to define two attributes: \texttt{memory\_bytes\_spilled} describes the amount of deserialized in-memory data before they are spilled to disk. On the other hand, \texttt{disk\_bytes\_spilled} represents the amount of serialized (compressed) data on disk after they are spilled.

For each stage with spill, Sparkscope calculates the \texttt{max\_memory\_usage} parameter as a maximum of \texttt{input\_bytes}, \texttt{output\_bytes}, \texttt{shuffle\_read\_bytes} and \texttt{shuffle\_write\_bytes}, representing the maximum needed usage of memory for the stage. After that, for each relevant stage, it calculates the \( \frac{\text{memory\_bytes\_spilled}}{\text{max\_memory\_usage}} \) ratio. If this reaches the threshold for low or high severity for at least one stage, a \texttt{StageDiskSpillMetric} object is created.

Besides the total amount of spilled data, Sparkscope is also able to display further details: for each stage, it displays the amount of spilled data (both memory and disk), also the \texttt{bytes\_read}, \texttt{bytes\_written}, \texttt{shuffle\_bytes\_read} and \texttt{shuffle\_bytes\_written} attributes to give the user a hint of what is the reason of the spill. Finally, it displays the task which is responsible for the largest part of the spill.

### 4.2.4 Driver or Executor Garbage Collector Heuristics

Sparkscope also investigates how driver or executors deal with garbage collection. For both, it calculates the \( \frac{\text{total\_gc\_time}}{\text{total\_duration}} \) ratio. After that, it checks if the ratio is too high or too low and if needed, a \texttt{DriverGcTimeMetric} or \texttt{ExecutorGcTimeMetric} object is created with the appropriate severity.
4. Implementation

4.2.5 Serializer Config Heuristic

As discussed in subsection 2.1.5, the Java serializer, which is used by Spark by default, is often not optimal. A preferred serializer (for instance, `org.apache.spark.serializer.KryoSerializer`) can be set in a configuration file. If a different serializer than the preferred one is used, Sparkscope detects an issue and `SerializerConfigMetric` object is created.

4.2.6 Dynamic Allocation Config Heuristic

In some cases, especially on shared or busy clusters, using dynamic allocation can help with sharing the resources more effectively. Therefore, Sparkscope provides an option to detect if the dynamic allocation is configured correctly.

The user can specify if the dynamic allocation is the preferred way of resource allocation (this can be done in a configuration file). If this is true, Sparkscope verifies if the `spark.dynamicAllocation.enabled` is set to `true`. If the dynamic allocation is activated for the application, Sparkscope also needs to check if shuffle tracking or shuffle service is configured correctly. This means that either `spark.dynamicAllocation.shuffleTracking.enabled` or `spark.shuffle.service.enabled` must be set to `true`.

Therefore, if the dynamic allocation is preferred, but not used, or not configured correctly, a `DynamicAllocationConfigMetric` with high severity is created.

4.2.7 Dynamic Allocation Min/Max Executors Heuristic

With dynamic allocation, the Spark application uses variable number of executors. The minimum and maximum possible number of executors that can be allocated can be set with `spark.dynamicAllocation.minExecutors` and `spark.dynamicAllocation.maxExecutors` properties in Spark. Analogously, the user can configure limits for minimum and maximum number of executors.

If the dynamic allocation is enabled, Sparkscope can compare the user defined value with the actual Spark configuration to avoid using more executors than would be necessary.

If the `spark.dynamicAllocation.maxExecutors` Spark property value is too large, or not set at all, the application can allocate too many executors or may even eat up all the cluster resources. This can be avoided either with setting up the maximum number of executors properly, or with proper configuration of the YARN resource pools and submitting the Spark application to the correct YARN queue. Therefore, Sparkscope raises an issue (represented by `DynamicAllocationMinMaxExecutorsMetric`) if the application is submitted to the default queue and if the `spark.dynamicAllocation.maxExecutors` exceeds the limit.
4.2.8 YARN Queue Heuristic

On clusters with multiple tenants, a frequent requirement is setting up resource pools in YARN, so that the teams or users of the cluster can get fair share of resources.

If necessary, Sparkscope can watch out for applications that are submitted to a default queue. The user can configure if the usage of the default queue is allowed and also what is the name of this default queue. If the default queue is not allowed, but is used, a YarnQueueMetric with high severity is created.

4.2.9 Memory Configuration Heuristic

Wrong handling of the memory is one of the most common causes of the bad performance of the Spark applications, or even of their failures. Unfortunately, unlike Spark Web UI, Spark History Server does not provide any details about the actual usage of memory (and therefore, neither does Sparkscope).

Therefore, only a simple configuration heuristic is used for detection of potential memory issues. As described in previous chapters, allocating either too much or too little memory can cause issues. Therefore, Sparkscope allows to set upper and lower limits for all four basic memory configuration parameters:

- Executor memory
- Executor memory overhead
- Driver memory
- Driver memory overhead

For each configuration parameter, Sparkscope uses four limit values – upper and lower limit for low severity and upper and lower limit for high severity.

If any of the four configuration parameters exceeds a limit, a MemoryConfigMetric is created, either with low or high severity. Furthermore, the Sparkscope GUI displays which parameters are responsible for raising the issue.

This heuristics should be interpreted very carefully, as Sparkscope has no information about how much memory is actually used by the driver or an executor. However, it might be helpful if the developer has good knowledge of the analyzed application and of the data, or it might help with a detection of extremely low or extremely high values in Spark configuration.

4.2.10 Core Number Heuristic

As discussed earlier, the optimal number of cores per executor or driver seems to be not an exact science. However, it is well known that the performance of either too small or too large executors if suboptimal. Also, some recommendations exist concerning the number of cores.
4. Implementation

Therefore, Sparkscope provides a heuristic for this. The four limits can be configured for the number of cores per executor or per driver: for each upper and lower limit for low and high severity are defined. If the Spark configuration exceeds any of the limit, the \texttt{CoreNumberMetric} object is created.

4.3 Presentation Layer

The web application and the routing to pages was written in Flask (version 1.1.2) \cite{flask}, one of the common web application frameworks for Python. The content of the forms was created with WTForms (version 2.3.3) \cite{wtforms} and Flask-WTF (version 0.14.2) \cite{flask_wtf} libraries. The graph carrying the Spark application history data was created using Plotly library \cite{plotly}.

The content of presentation layer itself was coded in HTML. For the variable content, Jinja2 templates were used to inject pieces of a Python code with a simple logic to the HTML files. The pages were styled using CSS and Bootstrap 4 library \cite{bootstrap}.

The presentation layer was coded based on the wireframes from section 3.3. Only small adjustments were made. The screenshots from the real application deployed on a test cluster are enclosed in Figure 4.2 and Figure 4.3.

4.4 Further Development

A working prototype of Sparkscope has been delivered. Nevertheless, I suggest several ideas for new features or improvements of the current application.

The first of them is finding another way how to fetch data from Spark. Even though the asynchronous execution was used, which helps to increase the performance, fetching data from a hundred Spark applications can take up to 10 minutes and generates a lot of HTTP traffic. Furthermore, the Spark History Server API does not provide data about memory usage and also seems to be buggy. During the development and testing of Sparkscope, I noticed that some of the endpoints do not provide consistent endpoints over time.

These flaws could be mitigated by using Spark event logs instead of History Server API. The event logs are files written by Spark to HDFS, which contain metadata about the application or its jobs, stages, tasks or executors. Using event logs could bring better performance, better chance to monitor the execution memory and ensure the consistency of the provided data. On the other hand, it would be definitely more complex from the development perspective, as it would require parsing the log files.

The second point is a reduction of the stored data amount. For the 20,000 Spark applications collected during the development and testing, the total data amount is about 15 GB. Out of those, about 85% are consumed by tasks, which are not so crucial for the heuristics that are in use so far. Therefore, it makes sense to consider reducing the data amount being processed.
4.4. Further Development

Figure 4.2: Sparkscope search page screenshot.
4. Implementation

Figure 4.3: Sparkscope compare page screenshot.
The current version of Sparkscope already provides an option to process and store only a limited number of the most demanding tasks within each application. Nevertheless, this optimization may eventually go further and omitting the individual tasks data might be considered. This could also contribute to improving the History Fetcher.

The third point for a possible improvement relates to the web application performance. The reconstructor described in section 4.2 is easy to use, but generates a lot of SQL queries which might be unnecessary and negatively affects the performance. There are more alternatives to this approach:

1. Implementing a caching layer that allows faster access to the data

2. Implementing Sparkscope as an ETL application with an entity describing processed applications. This could hold some detailed application metadata including the detected issues.

The fourth point regards to the user interface. The compare feature requires the user to enter the ID’s for each of the two application to compare, which is a bit clumsy and not very user friendly. Instead, some “add to basket” mechanism could be implemented.

Furthermore, the application history functionality could be extended to provide more information. So far, it only allows to monitor the application duration. Besides that, it might be worth it to keep track of more parameters – for example, the amount of the input or output data, the basic configuration parameters or detected issues.
Testing

Only a very small portion of code is covered by unit tests. This was the case for a few utility functions of History Fetcher.

The whole application was tested manually. Within the testing phase, History Fetcher ingested data from about 20,000 real applications running on a test cluster in an unnamed international company and saved it to a local database instance. Based on these database entries, the Web Application was also tested locally.

Furthermore, Sparkscope has been deployed on the test environment itself and it became accessible for the first set of users.

Also a simple test of the user interface was performed. Three users participated in the test.

User A is a software developer with ten years of experience with software development and five years of experience with development in Hadoop and Spark, and also with experience with the tuning of Spark applications and configuration of Spark clusters.

User B is also a software developer. He has been developing Spark applications for one year, but does not have much experience with tuning of the Spark applications.

User C is a non-IT professional with no knowledge of any programming language.

The Users A and B are considered to be typical users of Sparkscope. The User C will probably never use Sparkscope, but still can provide useful feedback to the user interface.

All the users were asked to perform the following actions in Sparkscope:

1. Find Spark applications which were started today by user X.

2. For the newest of these applications, try to find out:
5. Testing

- Which issues does the application suffer from?
- What is the value of the `spark.executor.memory` configuration parameter?
- How many tasks has the application executed?

3. Compare the applications with ID’s Y1 and Y2.

4. See the history of the application Z and see the details for the longest running instance.

All the users were able to fulfill all the tasks immediately, without any problems. This indicates that the user interface is intuitive and well understandable.
Conclusion

The presented thesis focused on a design and development of Sparkscope, an open-source application available on Github \[30\]. Sparkscope is a tool for support of performance tuning of applications written in Apache Spark.

Sparkscope is a tool consisting of two parts. The first part, called History F etcher, can retrieve data about performance of Spark applications from Spark History Server and store them in the relational database. The second part is the Web Application, which uses the saved data to analyze the performance of the Spark applications. As there are many factors having impact on the performance, several heuristics were derived and are used to evaluate the performance and detect potential issues. The results of the analyses are displayed in a user interface in a simple and concise fashion.

All the requirements specified in the thesis assignment have been accomplished. Nevertheless, on top of those, several points for possible improvements were identified, related to performance, new heuristics to calculate or a user interface improvements, elaborated in section 4.4. These will be become a subject of a further development.

Sparkscope prototype has already been deployed on a test cluster of an unnamed international company and analyzed more than 20,000 Spark applications.

Sparkscope does not necessarily have to replace the other optimization tools. Its purpose is to provide a quick overview of potential issues in Spark applications, an option to compare two applications side by side or to see a performance history of multiple instances of a recurring application. It can be combined with Spark History Server which can provide further details about a single application if necessary.

With Sparkscope, Spark developers might be able to tackle the performance issues of their applications more effectively. Therefore, they can reduce the workload and increase the throughput of the cluster, and hence, increase the revenue.
Bibliography


Installation Guide

Before the installation of Sparkscope, you need the following prerequisites:

- Python 3.6 installed
- pip, a package installer for Python
- PostgreSQL with admin console access
- Linux environment
- Running instance of Spark History Server

In order to install and start Sparkscope, you should do the following steps:

1. Get the Sparkscope files to the target location, either from the installation medium, or by cloning from Github with command
   `git clone https://github.com/stovicekjan/sparkscope.git`

2. Open the target directory (its subdirectories should be `db`, `history_fetcher`, `logger`, `sparkscope_web` and `test`)

3. Create Python virtual environment with the following command:
   `python3 -m venv venv`

4. Run the `setup.sh` script to initiate the configuration files and prepare the setup of the `PYTHONPATH`.

5. Activate venv with the following command:
   `source venv/bin/activate`

6. Adjust the `db/setup_db.sql` script (modify the password if necessary), and afterwards, run the script.

7. Edit the `db/db_config.ini` file and fill in the PostgreSQL password (use the value from the previous point) and the database hostname.
A. Installation Guide

8. Edit the `history_fetcher/config.ini` file and fill in `base_url` and `url` parameters, so that they point to your Spark History Server instance.

9. Edit the `history_fetcher/logger.conf` file. Inside the `handler_fileHandler` section, modify the `args` parameter and specify the path where the logs should be stored. You also need to create this path if it does not exist.

10. When the venv is activated, run the following command to install the necessary Python packages:
    ```bash
    pip install -r requirements.txt
    ```

11. Load an initial data with History Fetcher (see Appendix B for help)

12. Start Sparkscope web application with the following command:
    ```bash
    python3 sparkscope_web/main.py
    ```

Once Sparkscope is installed, you might also want to see and edit the `sparkscope_web/metrics/user_config.conf` file to adjust the thresholds for the heuristics. All the parameters are commented in order to provide a guidance.
History Fetcher User Guide

History Fetcher is a command-line based tool for fetching the data from Spark History Server API and storing it in a database.

The data can be fetched with the following command (applicable to Unix-like systems):

```
python3 history_fetcher/main.py [-h | --help] [--test-mode] [--truncate]
```

The following options can be chosen:

- `-h`, `help` for displaying help.

- `--test-mode` for running History Fetcher in a test mode, fetching only some fixed number of the newest application data (the number can be defined in `history_fetcher/config.ini` file as `apps_number` parameter). This can be convenient for testing or for an initial run of the application, as it can avoid fetching a large amount of data at once.

- `--truncate` for truncating all tables in the database before fetching new records.
Acronyms

**API**  Application Programming Interface  
**CPU**  Central Processing Unit  
**CSS**  Cascading Style Sheets  
**DAG**  Directed Acyclic Graph  
**ETL**  Extract, Transform, Load  
**HDFS**  Hadoop Distributed File System  
**HTML**  Hypertext Markup Language  
**GUI**  Graphical User Interface  
**ID**  Identifier  
**I/O**  Input/Output  
**JSON**  JavaScript Object Notation  
**JVM**  Java Virtual Machine  
**ORM**  Object Relational Mapping  
**OS**  Operating System  
**RDD**  Resilient Distributed Dataset  
**RDBMS**  Relational Database Management System  
**REST**  Representational State Transfer  
**SQL**  Structured Query Language
C. Acronyms

UI User Interface

URL Uniform Resource Locator

YARN Yet Another Resource Negotiator
Appendix D

Contents of enclosed DVD

- `readme.txt` ......................... the file with DVD contents description
- `sparkscope` ......................... the directory of source codes
- `thesis` ............................. the thesis text directory
- `_BP_Jan_Stovicek.pdf` ............... the thesis text in PDF format
- `_BP_Jan_Stovicek` .................. \LaTeX{} source code and other resources