Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

A Framework for Optimizing Spark Configurations

Daniel Balbach

Course of Study:	Mechatronik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Dr. rer. nat. Uwe Breitenbücher, M. Sc. Robin Pesl
Commenced:	April 7, 2022
Completed:	October 7, 2022

Abstract

The rising importance of data in modern life, industry, and society introduces a huge interest in processing them. Data-driven approaches nowadays are ubiquitous. Due to the increasing amount of data, the need to process large amounts of data has led to the development of complex, distributed, and scalable processing frameworks. Such a framework is the Apache Spark framework. It offers a rich set of functionalities like classic SQL analytics, machine learning functionalities, graph processing functionalities, and many more. However, the broad range of functionalities can potentially lead to problems. One of them is that due to the different requirement characteristics of the various Spark applications, the standard configuration of the Spark cluster may not be optimally adapted. A suboptimal configuration can lead to higher execution times or lower cluster throughput. Higher execution times can lead to higher costs in environments where the execution time is directly coupled to the billed costs, like in a cloud environment. Besides the financial aspect, a better-configured Spark application may also better use the provided resources and reduce the execution time, thus increasing the throughput. This work addresses this problem by designing and implementing an optimization framework for optimizing Spark configurations of a given Spark application. The optimization framework is then applied in a case study on two exemplary use cases using a Spark cluster in a Databricks environment of a private cloud to demonstrate its practicability. The results show the framework can optimize Spark configurations in general while causing only minimal effort to the applicant. However, outperforming the standard Spark configuration in the exemplary use cases proves to be challenging, especially due to observed runtime variances in the cloud environment. The distinction between statistical variance and real improvement is complex.

Kurzfassung

Die zunehmende Bedeutung von Daten in der heutigen Industrie und Gesellschaft hat ein großes Interesse an deren Auswertung zur Folge. Datengetriebene Ansätze sind heutzutage allgegenwärtig. Die steigende Menge an Daten, verbunden mit der Anforderung auch große Datenmengen zu verarbeiten, hat zur Entwicklung komplexer, verteilter und skalierbarer Frameworks zur Datenverarbeitung geführt. Ein solches Framework ist das Apache Spark Framework. Es bietet eine vielfältige Anzahl unterschiedlicher Funktionalitäten an, die von klassischen SQL Analysen über Maschinelles Lernen bis zum Prozessieren von Graph-Strukturen reichen, sowie zahlreichen weiteren Funktionalitäten. Dieses breite Spektrum an Funktionalitäten kann jedoch zu potentiellen Problemen führen. Eines davon ist, dass aufgrund der heterogenen Anforderungscharakteristika der unterschiedlichen Spark Anwendungen die Standard Konfiguration des Spark Clusters nicht optimal angepasst sein kann. Eine suboptimale Konfiguration kann bspw. zu höheren Laufzeiten oder geringerem Durchsatz des Clusters führen. Höhere Laufzeiten können zu höheren Betriebskosten führen, insbesondere in Umgebungen in denen die Laufzeit direkt mit den Betriebskosten verbunden ist, wie bspw. in einer Cloud Umgebung. Neben dem finanziellen Aspekt kann eine besser konfigurierte Spark Anwendung zur effizienteren Nutzung von Ressourcen führen, was in der Reduktion der Ausführungszeit resultieren kann und dadurch den Durchsatz erhöht. Diese Arbeit adressiert dieses Problem durch den Entwurf eines Frameworks zur Optimierung von Spark Konfigurationen einer gegebenen Spark Applikation. Das Optimierungsframework wird anschließend anhand zweier beispielhafter Anwendungsfälle evaluiert bei denen ein Spark Cluster in einer Databricks Umgebung einer Private Cloud verwendet wird, um dessen Praktikabilität zu untersuchen. Die erzielten Ergebnisse zeigen, dass das Framework generell in der Lage ist Spark Konfigurationen zu optimieren während nur minimaler Aufwand für den Anwender entsteht. Jedoch zeigt sich auch, dass im Falle der beispielhaften Anwendungen das Übertreffen der Standard Spark Konfiguration schwierig ist, insbesondere aufgrund der beobachteten Varianz der Ausführungszeiten in der Cloud Umgebung. Die Unterscheidung zwischen statistischer Varianz und einer tatsächlichen Verbesserung erweist sich als komplex.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI Artificial Intelligence. 15

API Application Interface. 19

ASAM Association for Standardization of Automation an Measuring Systems. 28

DAG Directed Acyclic Graph. 22

DE Differential Evolution. 9

DT Decision Tree. 30

ETL Extraction Transformation Load. 24

HDFS Hadoop Distributed Files System. 23

JSON JavaScript Object Notation. 38

JVM Java Virtual Machine. 20

MAPE-K Monitor, Analyzer, Planner, Executor, Knowledge Base. 7

MDF Measurement Data Format. 28

ML Machine Learning. 15

ORC Optimized Row Columnar. 62

PaaS Platform as a Service. 24

RDD Resilient Distributed Dataset. 7

REST Representational State Transfer. 24

RF Random Forest. 30

SQL Structured Query Language. 22

SVM Support Vector Machine. 30

UI User Interface. 24

UML Unified Modeling Language. 47

VM Virtual Machine. 44

1 Introduction

1.1 Background & Motivation

In modern society and industry, information in the form of data plays a huge role, and its importance is still rising. As a result, data sources are omnipresent and cover most areas of modern life.

Due to the proven success of algorithms in the area of Machine Learning (ML) and Artificial Intelligence (AI) [BME19; FKE+15; KKL10; MOC18] the benefit of processing large amounts of data has been recognized [ALA+19; JM15; XX14] not only to gain knowledge and an understanding but also to profit financially [ZSK15]. The value of processing data can range from understanding correlations to finding patterns or predicting new data. This effect has started a huge shift of interest toward processing large amounts of data. Companies nowadays are processing datasets in the range of petabytes daily using computing clusters consisting of hundreds of physical nodes [SDC+16; ZCS+18]. Studies show that the amount of data is still rapidly growing [AAh19].

The type of applications for which data serves as the base can range from performing analysis tasks to building powerful models in the area of ML trained by the provided datasets. Processing such vast amounts of data, however, creates new challenges. Classical centralized approaches to processing the data often show their limits as the provided hardware resources don't scale up to the required needs [ZSK15].

Using a cluster of computing nodes effectively is a requirement for processing large amounts of data. New, scalable frameworks that profit from increased computing resources have been developed and presented. They typically follow a distributed computing approach to avoid the hardware's physical limits. Due to the distributed approach, additional computing resources can be provided by adding further computing nodes. Examples of popular distributed big data frameworks are Apache Hadoop [PRGK14] and Apache Spark [ZCD+12]. Both frameworks have proven to be very successful for processing large amounts of data, and Apache Spark has now become the de-facto standard for distributed data processing [SBB20]. The growing amount of data [HCL21] underlines the need for scalable processing frameworks even further.

When the mentioned big data processing frameworks run in a cloud environment, the price a customer has to pay for the resources depends on the time the resources were rented. The costs connected to the rental time give great importance to using the cluster of computing nodes as short as possible to reduce the monetary expenses to a minimum. Several strategies are applicable to reduce the execution time of a Spark cluster.

Such strategies could be reducing the input data, optimizing the executed software to reduce the complexity of the application, or renting more computing nodes with increased resources to provide a larger amount of computing resources. However, all of those strategies come with severe

disadvantages. Reducing the amount of input data could lead to loss of information. Optimizing the executed software often requires deep knowledge about the processing frameworks that only domain experts have. Additionally, some operations may not be optimizable. Using a larger cluster of computing nodes may not be realizable quickly in the case of "on-premise" applications. In the case of renting resources in the cloud, it leads to higher costs.

Another approach that doesn't have the mentioned drawbacks is the possibility of achieving a shorter execution time by improving the configuration of hyperparameters belonging to the computing framework. The advantage of this approach is that no further computing resources are required, and everything can be left as it is. Optimizing the hyperparameters reduces the execution time by increasing the efficiency of the computing framework. The approach can reduce the costs due to the shortened execution time. This approach is very beneficial in lowering the spending costs for rented cloud resources. Recent studies have shown the potential success and improvements of tuning Spark configurations [BYL+18; CSG+18; CYW21; CYWL21; GT18; XHY22]. Many applications big data computing frameworks are used for are very heterogeneous. Each application may have particular requirements and characteristics since it is designed for a specific purpose. The heterogeneity makes it difficult to provide a generally applicable and optimal configuration. Hence the potential to increase efficiency by improving the configuration of hyperparameters exists.

However, improving the configuration of a distributed big data processing framework comes with complex challenges. In the case of Apache Spark, the framework has more than 180 configurable parameters [CYWL21]. Many of them do not influence the performance of the executed application, while some tend to have a strong influence on the performance [NMXS21; PGT17]. Using an inefficient Spark configuration can lead to significantly higher execution times and lower throughput of data [HCL21].

The high number of configurable parameters and the large value range of many numerical parameters make it difficult to choose and find an optimal configuration based on intuition and a limited number of trials. Therefore, a more efficient and systematic approach is needed to deal with the complex parameter space to find an improved, ideally close to optimum configuration.

This work addresses these challenges by presenting a framework for optimizing Spark configurations. The proposed framework proves its practicability when applied to practical use cases in a case study.

1.2 Problem Statement

Optimizing Spark configurations typically requires expert knowledge and is time-consuming. The necessary monitoring of long-running Spark applications contradicts a manual approach. Due to the different application scenarios that Spark supports and the rich set of functionalities that Spark offers, selecting the relevant configuration parameters is difficult. Furthermore, optimizing configurations using a manual approach is hardly parallelizable, increasing the effort even more. Deciding which configuration performed better than others may lead to a high effort, especially

for the statistical evaluation of the collected data. Therefore, systematic approaches would be beneficial. A manual decision on how the configuration should be further adapted, especially for a large parameter space, may lead to wrong choices in the case of a non-systematic approach.

The problems of optimizing Spark configurations in detail are:

- optimizing a configuration in a given, potentially high dimensional, parameter space is difficult in case of a manual approach,
- the correct format and values ranges of the different parameters need to be obeyed,
- deploying the Spark cluster with custom configurations requires expert knowledge,
- collecting relevant data for the evaluation of a configuration must happen during the runtime of Spark,
- evaluating the achieved results using different configurations requires a systematic approach.

All challenges and potential problems combined result in the fact that optimizing Spark configurations is typically done only by experts. For many application scenarios, an optimization of the Spark configuration is not considered due to the lack of such experts. A proposed solution should therefore be applicable also for non-experts. Reducing the need for Spark experts would allow lesser experienced Spark applicants to perform an optimization of the Spark configuration.

1.3 Objective of this Work

The purpose of this work is to design, implement and apply an optimization framework for optimizing Spark configurations in a Databricks cloud environment. The objectives in detail are

- investigating the current state-of-the-art Spark optimization techniques,
- designing a suitable Spark configuration optimization framework,
- implementing the optimization framework,
- applying the optimization framework in a case study,
- evaluating the achieved results.

2 Fundamentals

This chapter presents the most relevant technologies in the context of this work and describes them briefly. The chapter concludes with related work in the area of Spark configuration optimization.

2.1 Apache Spark

This section gives a brief overview of Spark, including the composition of a Spark cluster and its functionalities. This overview over Spark relies mainly on the works of [CZ18; SDC+16].

2.1.1 Overview

Apache Spark [Fou18] is a scalable, distributed, in-memory based, general-purpose data processing framework. It supports various potential application fields like structured data processing, ML, graph processing, and data stream processing [SDC+16]. Spark is implemented in Scala. The numerous Application Interfaces (APIs) realizations in different programming languages such as Java, Scala, Python, and R show the flexibility of Spark. Spark was created in 2009 by Matei Zaharia at the AMPLab, which is part of the University of California in Berkeley. The project became an open source project in 2010, and in 2013 it was donated to the Apache Software Foundation. Due to its rising popularity, more and more work was contributed to the project. As a result, spark nowadays has become the industry's de-facto standard for processing big data [SBB20].

A Spark cluster consists of [Fou22c]

- the driver program/driver node,
- the cluster manager,
- and at least one worker process.

In the driver program resides the "SparkContext" (Figure 2.1). It is the component that interacts with the application that is executed by the framework and is hence the component of the Spark framework that interacts with the user. It is the starting point of all executions in the Spark cluster. The SparkContext is configurable by the user upon its creation - an important aspect for this work since this is where configuration optimization occurs. Once initialized, the SparkContext connects to the cluster manager to allocate the resources specified in the configuration of the SparkContext. The cluster manager starts the executor processes on the worker nodes. The number of executors once again depends on the specified amount of resources specified in the configuration. More than

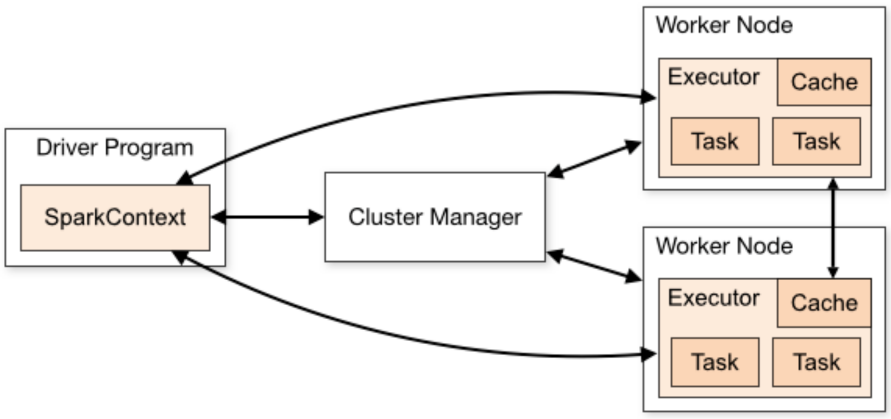

Figure 2.1: Overview over the different components of a Spark cluster [Fou22c].

one core per executor is possible. Each executor is a process and resides in a separate Java Virtual Machine (JVM). After starting the executor processes, they connect to the SparkContext in the driver program, and the cluster is ready to execute applications.

2.1.2 Resilient Distributed Datasets

RDDs [ZCD+12; ZCF+10] are one of the core components of the Spark framework. They are an important abstraction of the actual data that is processed. The attribute *distributed* in this context means that Spark distributes the data onto the cluster nodes. The degree of partitioning is adjustable. Adjusting the partitioning controls the degree of parallelization and whether the data is stored in memory or on disk. Therefore, the operations that are applied on the RDDs are executed in parallel. Since RDDs are immutable each applied operation on a RDD leads to the creation of new RDDs. Hence, the origin of a RDD is either other RDDs or the input data that is processed, as illustrated in Figure 2.2.

There are two distinct kinds of operations that can be applied on RDDs: “transformations” and “actions”. The group of transformation-operation contains all kinds of operations that create new RDDs. Examples of these operations are filtering a RDD or joining two RDDs. When using operations like filtering, each parent partition of the resulting RDD has at most one child partition. These transformations are called *narrow* transformations.

For operations like joining two RDDs, multiple partitions of the child RDD depend potentially on the same parent partition. These kinds of operations are called *wide* transformations [SDC+16].

“Actions” are operations that write results of executions to files or report the results back to the Spark driver. An essential aspect of actions is that only they trigger the actual execution of all operations. This behavior is called “lazy execution”. Once an action operation is requested, the distribution and order of transformations and the action operation itself are planned. This behavior allows optimizations since all the requested transformations are known to the framework. The

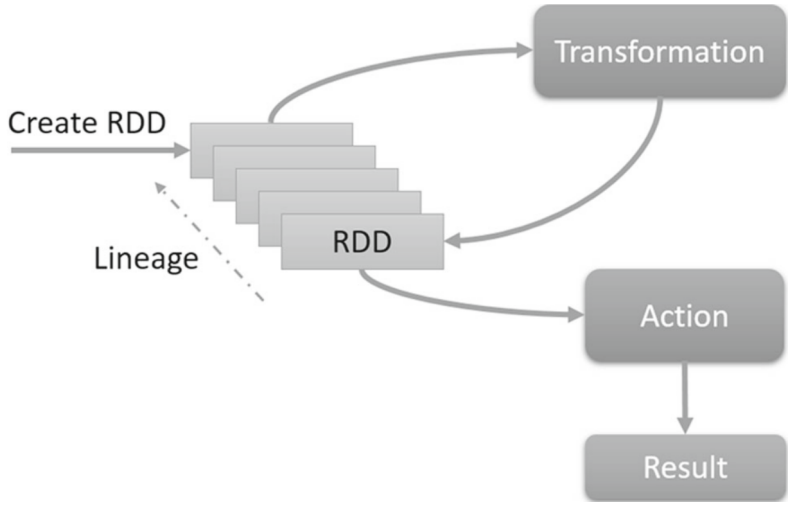

Figure 2.2: Overview over Transformations and Actions on RDDs [SDC+16].

plan of operations can then be optimized by applying optimizations similar to those of traditional database management systems like reordering or "predicate-pushdown". Finally, the executors execute the optimized execution plan.

The attribute *resilient* describes an important characteristic of RDDs. It means that they are fault-tolerant. Upon failure of cluster nodes, the Spark framework can recover the data. The resilience is realized by storing the execution plan that leads to the creation of the RDD. The operations are shifted to a working cluster node that takes over the faulty node's executions. This behavior is transparent to the user.

The actual execution can be separated into five hierarchical levels [ZLW18], namely

1. the application level,
2. the job level,
3. the stage level,
4. the task level and
5. the operation level.

The highest level is the application level. On this level, the user hands over the application to the SparkContext. The application may be a "Jar" or Python file for execution. The next lower level, the job level, contains all transformations that are predecessors of an action. Each job ends with the call of an operator that belongs to the actions group. An application can consist of multiple jobs.

The next lower level is the stage level. A stage is a collection of tasks that can be executed either sequentially or in parallel without shuffling. Shuffling describes the process of moving data between the executors. It is explained in detail later in this section. Shuffling marks the end of the current stage and lets a new stage begin. A job can contain multiple stages.

Tasks contain a set of operations that are applied on the partitions of a RDD. A task can consist of several operations as long as the operations only rely on a single RDD partition. Tasks are executed in parallel whenever possible if no (data) dependencies exist.

The lowest level of execution is the operation level. Multiple narrow operations are executed in one task as they don't rely on additional data input. Combining several narrow operations reduce the overhead of scheduling.

Spark builds a so-called Directed Acyclic Graph (DAG) to detect data dependencies. This DAG is built for each Spark job. Each node represents a stage, and each vertex represents a data dependency on one or more predecessor stages. The graph is a directed graph since it follows the specified operations of the job and ends with the execution of the action operation. The graph structure makes dependencies explicit. Stages that don't depend on each other can be executed in parallel. The creation of the graph before execution increases the degree of parallelization. Unlike previous big data frameworks like MapReduce, Spark can execute different types of operations in parallel due to the DAG, which increases the overall parallelization [LTW+15].

Whenever a data dependency in the DAG is detected, it is the trigger for Spark to perform shuffling before continuing the execution. Shuffling is called the process of redistributing the data between the different executors. Shuffling means data is physically moved in the cluster in one node or between different nodes using the connecting network. Due to the characteristics of network traffic this process is costly [CWL+20; HXLL17; ZCS+18]. For many applications, the part of the executed operations consuming the most time is shuffling [LTW+15]. Since many of the shuffling characteristics are configurable, the Spark configuration can have a great impact on the performance.

2.1.3 Spark Component Stack

The structure of the Spark framework consists of several layers. Upper layers use the provided functionalities of the lower layer. The highest level is the level that typically interacts with the user or developer. The highest level provides high-level functionalities and APIs, offering a convenient interaction for their users. Examples of such high level APIs are the "DataFrame" and "DataSet" APIs. These two APIs provide functions to process structured and semi-structured data. Due to the high-level context, they can offer a wide range of optimizations to reduce the computational cost of execution. Besides the two mentioned APIs, in this layer reside also the APIs for ML, data streaming and graph processing functionalities. Like the "DataSet", and "DataFrame" APIs, they provide high-level functionalities without having to know anything about internal implementation details or how to distribute the workload in the cluster.

The next lower level is the RDD API. For many application scenarios, it is sufficient to use the high-level APIs. However, special kinds of operations and application logic may not be supported by the high-level APIs. The low-level APIs of Spark can provide the solution for this problem.

However, using the low-level APIs may decrease performance. The reason for the decrease is the loss of performance optimizers that reside in the high-level APIs such as the "Catalyst Optimizer" that optimizes Structured Query Language (SQL) queries.

Additional features of the low level APIs are “Broadcast Variables” and “Accumulators”. Broadcast variables are used to share data efficiently across the cluster. They are immutable and can be used to provide additional data that might be needed to process a task. Using broadcast variables is more efficient than sharing standard variables in the Spark cluster that a programming language offers. In addition, broadcast variables avoid costly serialization and deserialization in each task.

Accumulators are mutable variables located in the driver process. Each executor can send updates to the accumulators reliably. Typical usages of accumulators are sums and counters. Broadcast variables and accumulators can be used to implement custom functionalities that the high-level APIs don’t offer.

The cluster managers are the next lower component below the Spark stack’s low-level APIs. The cluster manager is responsible for allocating the requested resources by the SparkContext. One of the reasons why Spark is very versatile is that it can use different cluster managers such as “Standalone” (Spark built-in cluster manager), “Hadoop YARN” and “Kubernetes” [Fou22c]. The support of different cluster managers allows the integration of Spark in already existing clusters and runtime environments.

The lowest level of the Spark component stack is the storage level. It is responsible for saving the executed transformations and buffering intermediate results if the provided memory size for the executors is insufficient or if an intermediate state of transformations should be preserved. Typically, distributed file systems are used in order to improve reliability and to offer faster writing and reading of files, such as “Hadoop Distributed Files System (HDFS)” [Bor22], “Cassandra” [Fou22a] or “MongoDB” [Mon22]. The support of the various storage systems underlines the versatility of the framework.

2.1.4 Spark Configuration

Spark comes with many adjustable settings and configurations due to the complexity of Spark [Fou22d] to control its behavior. Version `3.3.0` of Spark has more than 300 adjustable configuration parameters [Fou22d]. The parameters can be divided into 16 different groups, *“which are application properties, runtime environment, shuffle behavior, Spark UI, compression and serialization, memory management, execution behavior, networking, scheduling, dynamic allocation, streaming, SparkR, deploy”* [CYWL21].

Only Some parameters are adaptable at runtime.

Many parameters do not influence the execution of Spark applications like the application name (`spark.app.name`). However, some parameters are closely related to the execution behavior of a Spark application, like the number of executors (`spark.executor.instances`) or the provided memory for the driver process (`spark.driver.memory`). For many parameters, their influence and the effect of interaction with other configuration parameters are not evident. The lack of knowledge makes it difficult for Spark users to choose an optimal configuration or to tune the default configuration [Fou22e]. Additionally, the different parameters form a high dimensional space with infinite possible configurations. Therefore, optimizing this large set of parameters requires either expert knowledge or a systematic approach.

2.2 Databricks

Databricks [Dat22a] is a cloud-based Platform as a Service (PaaS) framework. Due to the strong focus of Databricks on data and analytics, Databricks provides extensive functionalities regarding ML and AI. Therefore Databricks offers a range of features for data warehousing like Extraction Transformation Load (ETL) processing, data ingestion, and data management. Databricks offers tools to support users to easily build data processing infrastructures.

Databricks is tightly coupled with Spark since the inventors of Spark founded the Databricks company [Ili20]. Databricks is responsible for configuring a cluster and for the automated deployment of Spark on the cluster. It enables its users to conveniently use Spark without needing know-how about technical deployment. Databricks provides many configuration settings to configure Spark and the cluster on which Spark is running.

Databricks allows the deployment of custom libraries and software on the cluster conveniently. The user of Databricks can hence execute any custom Spark application and business logic in a scalable cloud environment.

One of the core functionalities of Databricks are so-called “Databricks Notebooks”. They allow users to write and execute code interactively in languages like Scala and Python, similar to “Jupyter Notebooks” [Jup22]. In addition, mixing the programming languages in separate code cells is possible.

All functionalities of Databricks are accessible via a web-based User Interface (UI). In this UI, new clusters can be configured and launched, running clusters and jobs can be monitored as well as the state of Spark on the clusters itself. Additionally, ETL pipelines and tables can be built, ML models can be trained, and code for notebooks can be written interactively. However, Databricks offers many more functionalities, and new functions and tools are added frequently.

Apart from the access to Databricks via the web UI, the Databricks platform can be also accessed via a powerful Representational State Transfer (REST) API [Dat22b]. The API makes the access to the Databricks platform independent and allows to interact with Databricks programmatically too, which ultimately offers a way to interact with a Spark cluster.

2.3 Self Adaptive Systems

One desirable goal of software systems is the ability to adapt to changes during the runtime that occur in the runtime environment. This property makes such kinds of systems more robust and flexible due to their ability to react to changes. This type of system is called “self-adaptive system”. Two main functional parts can be distinguished in self-adaptive systems [Wey21], namely the managing and the managed system. The managed system deals with the domain in which the system is located, and the managing system deals with the system’s adaption.

Self-adaptive systems consist of four different elements [Wey21], namely environment, managed system, feedback loop, and adaptation goals, as illustrated in Figure 2.3. Together with the feedback loop, the adaption goals represent the managing system. It is the responsibility of the managing

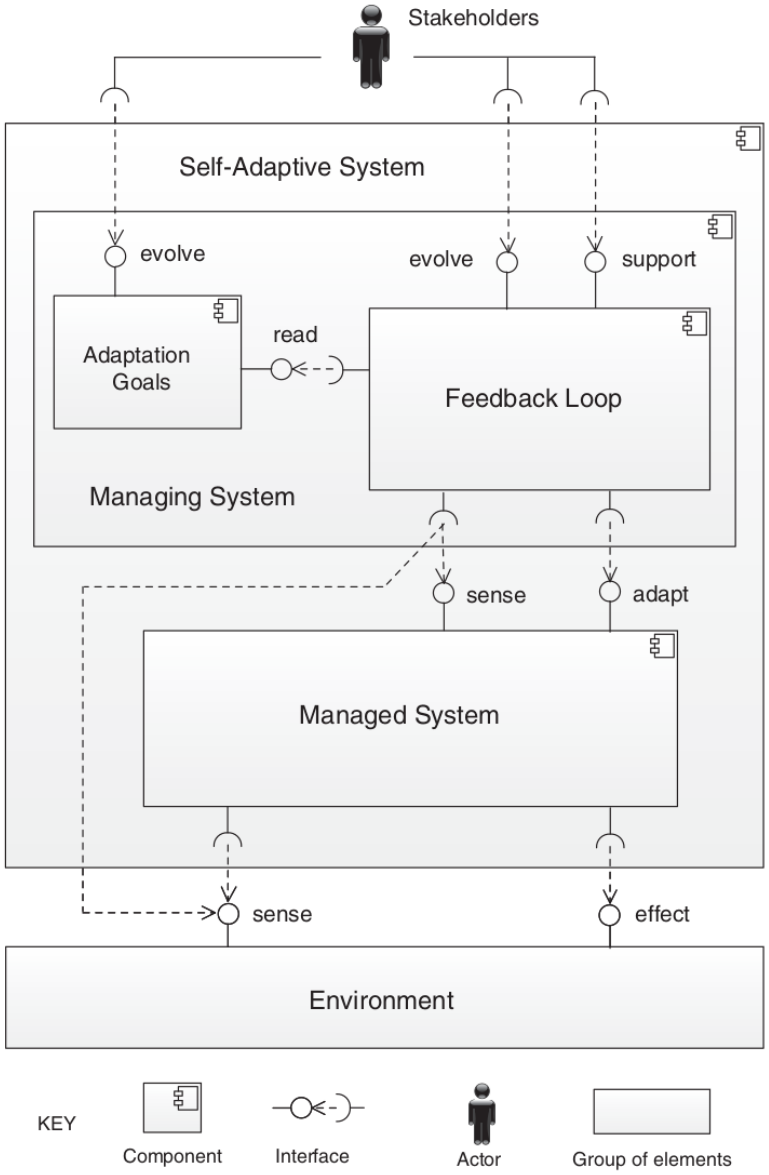

Figure 2.3: Reference model of self-adaptive systems [Wey21].

system to detect whenever changes in the managed system are necessary to fulfill the specified adaption goals. Hence, the managing system must be able to observe and adapt the managed systems by using sensors and actors.

The reference model of the managing system is the MAPE-K model [Wey21]. It consists of the five components *Monitor*, *Analyzer*, *Planner*, *Executor* and the *Knowledge Base*. All components meet their decisions and perform actions based on the knowledge contained in the knowledge base that is available to every component. Every component can update or retrieve the newest knowledge.

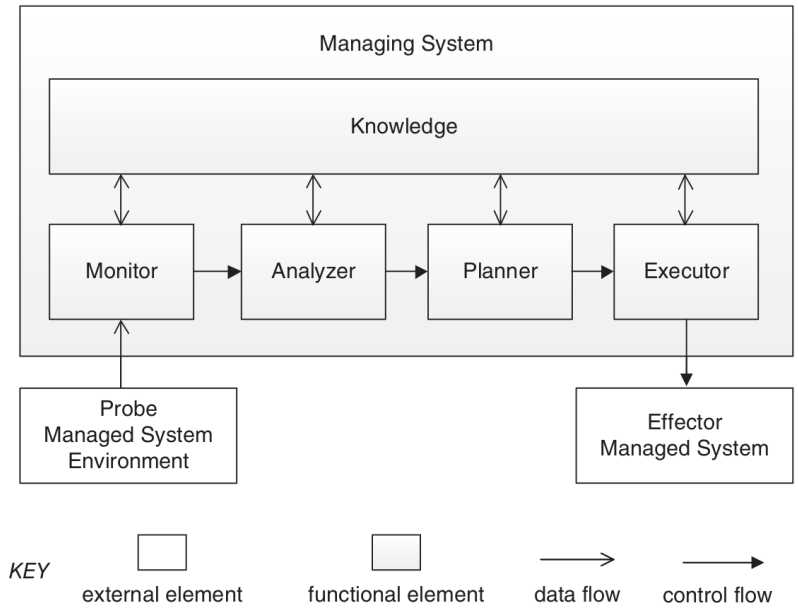

Figure 2.4: MAPE-K reference model [Wey21].

The idea of the MAPE-K reference model is a sequential workflow (Figure 2.4), beginning with the *Monitor* component. Its purpose is to monitor the managed system and to collect relevant data. The purpose of the *Analyzer* component is to decide whether an adaption of the managed systems is necessary. If this is the case, the *Planner* component plans the steps of actions that are required for the adaption. Finally, the *Executor* component executes the plan of actions for the adaption. A change in the managed system will trigger a new execution of the described workflow again.

The described reference model serves as a reference architecture for the optimization framework presented in this work. Finding an optimized Spark configuration can be seen as a self-adaptive system that continuously searches for improved configurations. The framework's purpose (managing system) is to react to the observed performance of proposed configurations for the Spark cluster (managed system). Depending on the specified optimization goals (adaption goals), the optimization framework's task is to suggest a new and better adapted Spark configuration.

2.4 Differential Evolution

Differential Evolution [SP97] is a "*parallel direct search method*" that is used to optimize a set of parameters belonging to a continuous, multidimensional parameter space. A cost function evaluates the proposed parameter values and makes hence the proposed sets of parameter values comparable. The goal is to find the set of parameter values that cause the lowest costs. One of the strengths of DE is that it can handle nonlinear, gradient-free optimization problems.

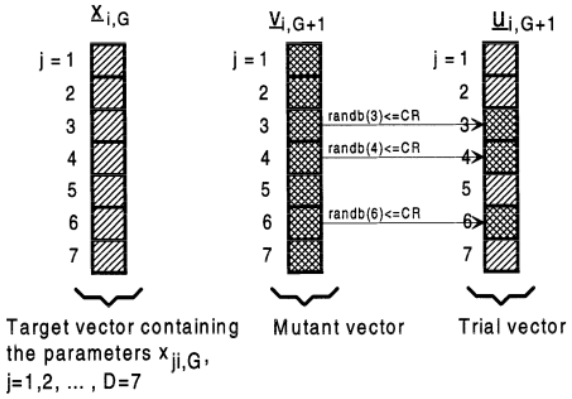

Figure 2.5: Exemplary applied crossover methodology [SP97].

The optimization process starts by generating a set of random vectors representing the so-called population. Varying each population candidate is called *mutation*. A mutation is achieved by building a weighted difference between two members of the population and adding the difference to a third member, as shown in Equation (2.1), which forms the potential new population member of the next population generation $G + 1$. The weight F controls the impact of the difference.

$$v_{i,G+1} = x_{r_1,G} + F \cdot \left(x_{r_2,G} - x_{r_3,G}\right) \tag{2.1}$$

The authors propose an additional method to introduce further variance which is called *crossover*. Its mathematical formulation describes Equation (2.2). Crossover controls how many of the mutated parameter values are accepted. A parameter is accepted when a sample drawn from a uniform distribution exceeds the specified threshold CR. In case of rejection, the parameter values of the vector from the previous generation $x_{ji,G}$ are reused. This process is repeated for every dimension j of all available dimensions D.

$$u_{ji,G+1} = \begin{cases} v_{ji,G+1} & \text{if}(randb(j) \leq CR) or j = rnbr(i), \\ x_{ji,G} & \text{if}(randb(j) > CR) and j \neq rnbr(i) \end{cases} \quad j = 1, 2, ..., D \tag{2.2}$$

Figure 2.5 illustrates the crossover-methodology.

The cost function tests and evaluates the resulting new population member. If the new population member achieves a lower cost value, it replaces the old population member. Otherwise, it is rejected. Evaluating the different proposed population members can be done in parallel and independent of each other.

The described properties make the DE algorithm a promising optimization algorithm for optimizing Spark configurations.

2.5 Measurement Data Format

The Measurement Data Format (MDF) format is a proprietary data format that follows the specifications issued by the Association for Standardization of Automation an Measuring Systems (ASAM) association. The authors of the MDF format describe it "*as a compact binary format, ASAM MDF offers efficient and high performance storage of huge amounts of measurement data. MDF is organized in loosely coupled binary blocks for flexible and high performance writing and reading. Fast index-based access to each sample can be achieved by loss-free re-organization (i.e. sorting) of the data. Distributed data blocks even make it possible to directly write sorted MDF files. The file format allows storage of raw measurement values and corresponding conversion formulas; therefore raw data can still be interpreted correctly and evaluated by post-processing tools*" [Bas22].

Among many other fields of applications for measurement data, one field of application is the area of automotive measurement data which is the field of appliaction in this work. Due to the development of new automotive systems like information and entertainment systems, autonomous driving and assistance systems, the amount of data in an automotive system is increasing fast.

Processing such large amounts of measurement data is key for data-driven development approaches. However, the large amount of data requires scalable processing strategies. Since Spark offers generic data processing capabilities, Spark can be used for binary data processing, like in the case of the introduced MDF format.

2.6 Related Work

Tuning the Spark configuration to achieve better performance was already subject to past publications. In this section, relevant and related works are briefly presented.

In [PPCB20], the authors use a black box approach to model the behavior of an unknown objective function, like the execution time of a Spark application. To optimize this function, like minimizing the execution time of an application, they use the "Bayesian Optimization" technique. This technique is used when a large, high-dimensional search space must be examined efficiently. The input into the function is composed of the Spark configuration and additional information like the dataset size. Since most of the spark applications are very heterogeneous in their usage of the provided resources, the authors propose a method to encounter this problem by using parsed "SparkEventLogs". The logs contain information like CPU usage, I/O usage, shuffling characteristics, garbage collection time, and information about how the application was executed, like the operations in the stages and tasks. The spark configuration, enriched with the additional features, serves as input data for a machine learning-based model. The model is trained with the results of an initial set of runs with different Spark configurations. Then the model is used as an oracle that predicts the execution time of the Spark application, given a specific Spark configuration. The oracle is queried using the Bayesian Optimization strategy until a good configuration is found. The authors could demonstrate that a single execution of a new and unseen Spark application was enough to find a significantly improved Spark configuration by querying the oracle and providing the SparkEventLogs.

In [NMXS21], the authors aim to improve the Spark configuration for known and unknown applications. First, they begin to search for the relevant parameters regarding the execution time by conducting many experiments with different Spark configurations and therefore include all relevant parameters that might influence the performance. Hence no parameter is filtered out a priori. The pertinent parameters affecting the performance are then selected using a statistical hypothesis test. Thus no parameter is overseen or previously filtered out due to false assumptions about its performance. Having selected a subset of relevant parameters, the authors then execute various Spark applications with the default Spark configuration and track the behavior of the cluster that runs the application. The observed characteristics consist of, among other things, CPU usage, I/O, IPC, and cache behavior. The collected information is then used to characterize the executed application. Next, the collected information is then resampled and transformed to use the transformed information for clustering algorithms. Then, a KMEANS algorithm clusters the data. The cluster then represents the characteristics of the application regarding its system usage in an abstract manner. For each cluster, various ML based performance models are trained and evaluated. As input data, the models receive the Spark configuration, the data size, and the mean values of the observed resource usage. The authors show that a Random Forest Model achieves the best prediction results.

Furthermore, the authors demonstrate that it is sufficient for new, unknown applications to run the novel application only once, using the default Spark configuration and recording the system's resource usage. Based on the resulting cluster, the relevant performance prediction model can be used to find the optimal Spark configuration. The authors demonstrate that this approach can successfully improve the performance of known and unknown Spark applications.

Predicting the execution time of Spark applications efficiently is also the goal of [CYWL21]. In this work, the authors aim to optimize the Spark configuration to achieve a lower execution runtime or higher throughput. Therefore they model the execution time for each Spark job, its containing stages, and their scheduling time. For each type of Spark job and for every kind of its containing stages, a regression-based approach predicts the execution time of the stage. To solve the regression problem, the "Adaboost" algorithm is used. It is an ensemble learner that trains multiple underlying models for either regression or classification-based problems. The Adaboost algorithm is then used to train numerous regressors that predict the execution time of the specific stage based on the given Spark configuration. The advantage of this ensemble model approach is that it's robust to overfitting and can deal with complex relationships in the training data. With the performance models of each type of stage, the whole execution time of a Spark application can then be calculated based on the function that models the entire execution time of the application.

To minimize the required training samples, the authors present the approach of "projective sampling". This approach tries to infer the relationship between the number of training samples and the resulting prediction error of the model. Then a cost function is used that considers the error of a predicting model and the cost of achieving new samples. Projective sampling gives the ability to determine the optimal sample size needed to achieve the highest model accuracy at the lowest amount of samples possible. This approach avoids unnecessary and costly executions of Spark applications to collect training data. The authors were able to show that their approach was able to achieve a higher prediction accuracy by using the Adaboost approach, contrary to traditional ML based

approaches like Support Vector Machines (SVMs) or Random Forest (RF). They could further show that their approach needs fewer training samples simultaneously to predict the execution time of various workloads and applications.

Reducing the execution time and the execution costs is the goal of the proposed framework in [CYW21]. To achieve this goal, the authors consider not only optimizing the Spark configuration to achieve a lower execution time for a given application. The authors also consider the costs that need to be paid in public cloud environments. A multi-objective optimization framework is therefore proposed. The core part of the framework is the "Configuration Searcher". It is responsible for finding optimized configurations that reduce the execution time and simultaneously keep the used resources' costs low. This part of the framework consists of an execution time prediction model based on the "Adaboost" algorithm, which trains several different Decision Trees (DTs) to solve the regression problem. This approach is combined with a genetic algorithm that selects the following configurations to be evaluated by the model based on previous successful configurations. To evaluate the configurations, the authors present an objective function that considers the execution time and costs and serves as the base of the optimization. The authors show that their approach outperformed a modern, general-purpose optimization framework in various applications and workloads. Furthermore, the execution times and costs were substantially lower than those of the general purpose optimization framework. Due to the extensible nature of the presented framework, it is also highly adaptable and can be easily extended for future new Spark parameters.

The approach of [CSG+18] is to enrich a black box model with additional information about the execution of Spark applications. The authors follow a gray box modeling approach. The proposed approach is to build performance models per stage level that can predict the execution time of a stage for a given Spark application. The added information to the black box approach is, among other things, the provided cluster resources, the potential parallelism due to the provided resources, and the history of past executions in the form of logs. Then several different regression algorithms are applied to predict the execution time of a stage. Among them, the best-performing algorithm is chosen. As a final step, the expected execution times of all stages are aggregated. Hence, the selected model can forecast the total execution time of the application. The authors demonstrated that this approach received good results in predicting the execution time for different applications.

In [PGT17], the authors use a manual approach to investigate the influence of Spark configuration parameters on the resulting execution time. Based on manual prefiltering of relevant configuration parameters in Spark, the authors perform multiple experiments varying one parameter at a time. This process is repeated for all parameters of interest and various kinds of workloads and applications. The most considerable influence for almost all types of workloads that the authors observed was the choice of the serializer. By using the "KryoSerializer" the authors observed a massive decrease in the execution time. Another important observation from their experiments was that not all parameters were equally relevant for different applications. From the results of the experiments and the resulting insights into the impact of specific parameters on the performance, the authors propose a simple methodology for tuning the Spark configuration. The main idea of this methodology is to focus on the parameters individually. The influence of each parameter is then tested in an experimental run. When a positive impact on the performance is observed, then the parameter is kept. Otherwise, the default configuration is kept. In the end, based on the results, one Spark configuration is assembled. To reduce the number of parameters that need to be investigated, the

methodology focuses on those parameters where the authors could observe the highest impact on the performance. Thus, they effectively reduce the number of experiments that need to be performed to only 10.

The authors of [LTW+15] present a framework for performance evaluation and comparison. Therefore the authors use different workloads for the different application fields of Spark like SQL, ML, Graph processing, and data streaming. A beneficial property of the framework is the generation of synthetical benchmark data via the Spark "Data Generator". The authors observed two crucial aspects. One was the large number of Spark tasks that required shuffling. Hence shuffling performance is critical for the execution of the workloads due to its costly nature. The other observation was that for most of the various applications, the available memory of the executors is of great importance due to the concept of RDDs as they reside in the memory of each executor.

The work of [XHY22] proposes a framework for optimizing Spark-SQL queries consisting of several main components. The name of the framework is "LOCAT", meaning "low overhead Online Configuration Auto-Tuning of Spark SQL Applications". First, they propose an analyzing component to evaluate whether the query is sensitive to parameter tuning. Having identified the queries sensitive to parameter tuning, they then introduce a second component to identify relevant configuration parameters. The set of relevant configuration parameters then serves as input into a third component that uses an adapted Bayesian-Optimization optimizer to find the configuration that results in the lowest execution time. The particularity of their optimizer is that it also considers the data size. Finally, the authors demonstrate the success of their work on several benchmarks. The overhead of optimization using their framework is significantly lower compared to other optimization frameworks. Due to the consideration of the input data size, the optimized configurations can be used for different input data sizes.

In [HCL21], the authors present a survey of different optimization strategies for parameter tuning of big data processing systems. They mainly focus on systems for batch and stream data processing. The various optimization strategies are grouped into six categories: rule-based, cost modeling, simulation-based, experiment-driven, machine learning, and adaptive. For each group, an in-depth review is provided with its respective advantages and disadvantages.

3 Concept and Architecture of the Framework

This chapter describes the concept and the architecture of the proposed optimization framework. The solution is derived to address the problems specified in Section 1.2.

3.1 Optimization Goal

The mathematical formulation of the goal of optimization can be formulated as stated in Equation (3.1) [HHL18]. The performance modeling function F models the Spark cluster with the provided configuration $conf$. The term $perf$ describes the performance metric. Performance metrics may be the total execution time, response latency, or execution costs. The modeling function F consists of several parameters that influence the target value that should be optimized. Factors that influence the performance metric subject to the optimization are the application app that processes the input data and the amount of input data represented by the $data$ term. The resource's configuration res is, in the case of this work, the configuration of the Spark cluster's nodes. The goal of the optimization is to find a Spark configuration $conf^*$ that minimizes modeling function F, as shown in Equation (3.2). The task of the optimizer is finding an optimal or close to optimal configuration regarding the chosen metric of interest.

$$perf = F(app, data, res, conf) \tag{3.1}$$

$$conf^* = \underset{conf}{\operatorname{argmin}} F(app, data, res, conf) \tag{3.2}$$

3.2 Optimization Approach

One challenge in optimizing the configuration of a spark application is that each Spark application behaves differently. It may use different functions of the Spark APIs, as shown in Section 2.1. As a consequence, each parameter of the configuration might have a different impact on the respective application. The parameters' potential impact may range from no impact to a high

impact. Additionally, the amount of processed data, its format, and its structure may influence which Spark configuration parameters may be important. A “black-box” approach for modeling the Spark application is chosen to handle this lack of knowledge.

Using a black box approach means that there is no modeling function of the Spark cluster itself. The Spark cluster’s behavior reacting to different inputs must be observed. The advantage of a black box approach is that the complexity of the Spark cluster doesn’t need to be modeled at all; the correlations of input and output must be only observed. This empirical approach has the advantage of covering all characteristics during runtime. It is applicable for all kinds of Spark applications. A black box approach also handles the lack of knowledge about the influence and correlation of influences for Spark configuration parameters since it makes no assumptions about any correlation.

However, it also has disadvantages. Since no assumptions are made about the Spark cluster, there is no other way than empirically trying out how the Spark cluster reacts to different sets of configurations.

Many applications, especially in the area of processing big data, can cause high costs due to long runtimes. This is why searching for new configurations should be as efficient as possible to reduce the effort of the empirical approach.

3.3 Framework Conception

For the optimization of Spark configurations, the optimization framework needs to support various functionalities such as

- providing the functionality to specify a parameter space in which the configurations are located,
- searching the parameter space and proposing new configurations that should be investigated,
- specifying an optimization function that for each investigated configuration evaluates the behavior of the execution and thus makes them objectively comparable,
- executing the Spark applications with the proposed configurations in order to empirically observe their impact on the behavior of the application,
- observe and collect relevant statistics during the execution of the Spark application for the evaluation of the impact of a certain configuration,
- evaluating the results of the application’s execution,
- proposing new, improved configuration candidates based on the empirical results of the investigated configurations.

With these specifications, a concept of the framework can be derived and implemented. One challenge in optimizing the configuration of a Spark application is that each application behaves differently. It may use different functions of the Spark API (compare Section 2.1), and therefore each parameter of the configuration might have a different impact on the respective application. The potential impact of the configuration parameters may reach from no impact to a high impact.

Given the high number of adjustable Spark parameters [Fou22d], the parameter space of potential Spark configurations is high dimensional, and an infinite number of parameter value combinations exist. This makes finding optimized configurations complex and challenging. Strategies like an exhaustive search to find optimized configurations are neither feasible nor possible. This is why the framework presented in this thesis uses a more efficient approach to find and propose optimized configuration candidates. Inspired by the work of [CYW21], the used search algorithm belongs to the group of "genetic algorithms" [SP97].

The DE algorithm fits well to the framework's requirements as it's gradient-free and can handle large, high-dimensional search spaces [SP97]. In Section 3.4.1, the adaptions of the DE algorithm are described.

For the specifications of the parameter space, a domain-wise approach is selected. This means that each domain has its own parameter space that can be searched and taken into account for optimization. The motivation for this approach is that several optimizable domains in the Spark stack already exist, such as application settings, Spark settings, and cluster settings. Any additional domain is possible. In this case, the domain-wise optimization approach makes it possible to optimize parameters for each of the domains, namely the cluster configuration, the Spark configuration, and the Spark application's configuration. Also, additional domains might be required for the optimization of future applications to handle special requirements. The flexible and modular approach of the domain-wise parameter space specifications makes it possible to deal with these requirements.

For each domain, the parameter space is specified separately. The optimization framework offers three major datatypes to describe a parameter's datatype. The datatypes are integers, floats, and categorical string values. In the case of the Spark stack, the three datatypes are enough to model all datatypes of parameters. The parameter specification then serves as the base to model the parameter space in which an optimized configuration is searched.

Searching the parameter space and proposing new configurations is done by the core component of the optimization framework; the *ConfigBuilder*. The purpose of the ConfigBuilder is to propose new, improved configurations based on the results of the performance of previously examined configurations evaluated by the given optimization function. The ConfigBuilder in detail is described in Section 3.4.1 and Section 4.2.

Once the ConfigBuilder proposes a set of configuration candidates, this set is then evaluated empirically. Before executing the application with the respective configuration, the corresponding resources must be configured and provided. In the case of a Spark application, this means configuring a cluster of physical computing nodes and deploying the Spark framework on the cluster. Once the Spark cluster is ready for execution, the actual Spark application is executed.

In the case of this work, setting up the cluster and deploying the Spark framework on the provided resources is taken over by the Databricks platform (Section 2.2). Among other things, Databricks offers PaaS. In the provided runtime environment, the Spark framework is already set up and configured and can be directly used by the user. The configuration for the cluster of computing nodes, the Spark configuration itself, and user-defined configuration parameters for the execution of the Spark application can be specified via the Databricks REST API. This makes executing applications with different settings in the application stack convenient. However, it requires a dedicated component that interacts with the REST API.

After the execution of the Spark application with the given configuration candidates, the results of the execution need to be evaluated and assessed by the specified optimization function. Depending on the optimization goal, different collected information and execution statistics need to be investigated and assessed. The results are forwarded to the ConfigBuilder afterward. The ConfigBuilder then proposes new and potentially better configuration candidates that should be elaborated in the next iteration. Thus a feedback loop is realized, similar to the one specified in the MAPE-K reference model (Figure 2.3).

The search of potential optimized configuration candidates is hence not random but uses the information on the performance of the previously investigated configurations. Compared to a random search, this approach is more efficient.

By closing the loop, the optimization becomes an iterative process. The advantage of an iterative approach is that the process can be stopped after any iteration; on the other hand, it can be continued at any time. This allows the search for optimized configurations to be halted once a sufficiently improved configuration is found. Then, at a later point in time, the search for an improved configuration can be continued.

3.3.1 Optimization Workflow

The underlying procedure of optimizing a configuration is illustrated in Figure 3.1. In the beginning, the configuration of the search algorithm must be specified. The configuration determine, among other things, how many configuration candidates should be proposed per iteration. Additionally, the domain-wise parameter specifications must be handed over to the ConfigBuilder. For each parameter specification, its name, datatype, and the value range of the potential parameter values must be specified. With the specified information, the ConfigBuilder can be instantiated. Depending on the specified configuration settings, the ConfigBuilder suggests a set of configuration candidates consisting of the specified number of configurations. The influence of every configuration candidate is then investigated empirically. This means the configuration is deployed on a real Spark cluster, and the Spark application is executed with the given configuration. Since the Spark cluster in a cloud environment is typically hosted on virtual machines, it may be subject to external influences.

Such influences could be the resource usage of other customers that use the same physical resources. Therefore, it is mandatory to try out each configuration several times. Thus, it is avoided that single outliers dominate the optimization process, which aren't caused by the change of configuration settings but by external factors.

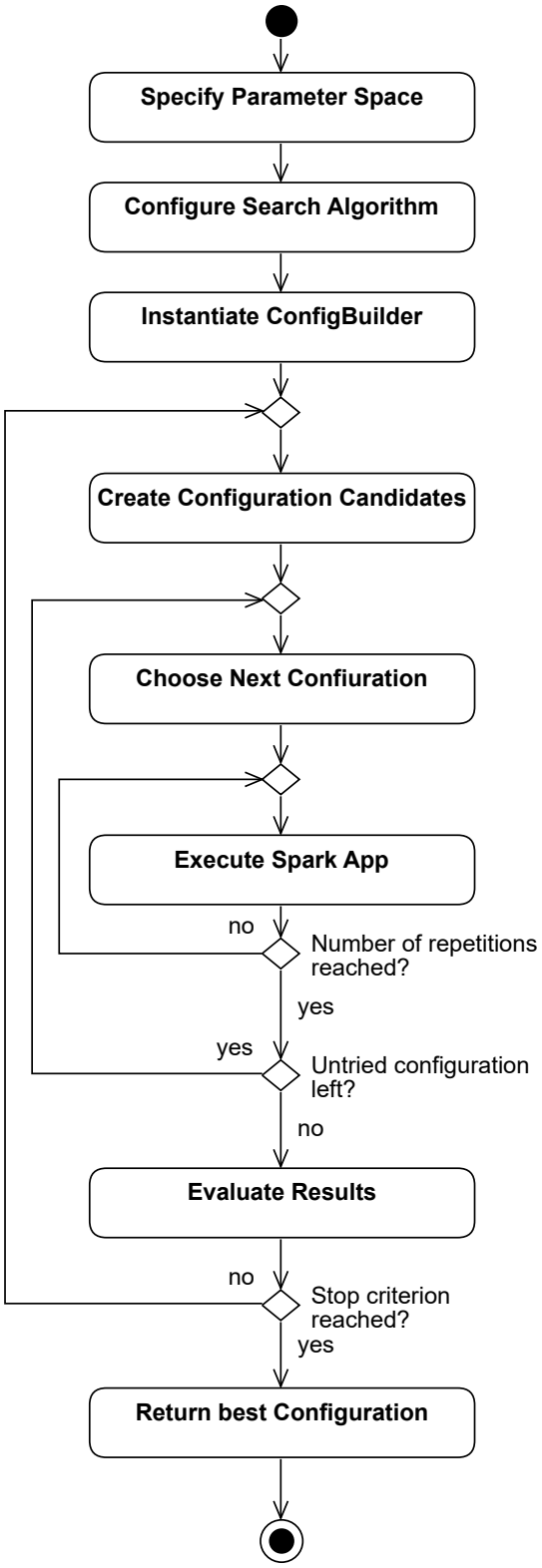

Figure 3.1: Flowchart of the configuration optimization procedure.

The problem with outliers is that it is unclear whether they occur due to the chosen configuration or due to other factors. Such factors could be a temporary high usage of the shared hardware resources by other applications that run on the same hardware infrastructure, like the Spark cluster. Other factors could be a transient high load on the network or the storage servers. Without several repetitions, distinguishing whether the performance is related to the configuration or external factors is impossible.

Once the specified number of repetitions is reached, the different runs can be aggregated using a specified aggregation function. This procedure is applied to every configuration candidate until all candidates in the set of configuration candidates are investigated empirically.

The next step is to evaluate the results of each configuration candidate. Based on the achieved results, the optimization is either stopped, and the best found configuration is returned, or the next iteration of the optimization begins. This means that the ConfigBuilder instance is informed about the achieved results of the configuration candidates. Based on that information, the ConfigBuilder then proposes a new set of configuration candidates, and the previously described procedure starts again until the optimization loop is stopped.

3.4 Framework Architecture

The iterative cycle described in Section 3.3.1 and illustrated in Figure 3.1 can be recognized also in the architectural overview in Figure 3.2. The core component of the framework is the ConfigBuilder. As previously described, it is responsible for proposing new configurations for the investigation. Since the ConfigBuilder knows nothing about the target runtime environment and how to deploy and run the actual Spark applications, an additional component is required — the “ExperimentRunner”. Its purpose is to transform the proposed configurations into valid configurations for the target environment. This includes the format itself, i.e., how the parameters are structured, and deploying the Spark application.

In the context of this work, adapting the proposed configuration means transforming the parameters into a JavaScript Object Notation (JSON) format that corresponds to the specifications of the Databricks API [Dat22b]. The ExperimentRunner component also controls how many different experiment runs are executed in parallel, monitors their execution, and handles errors if they occur.

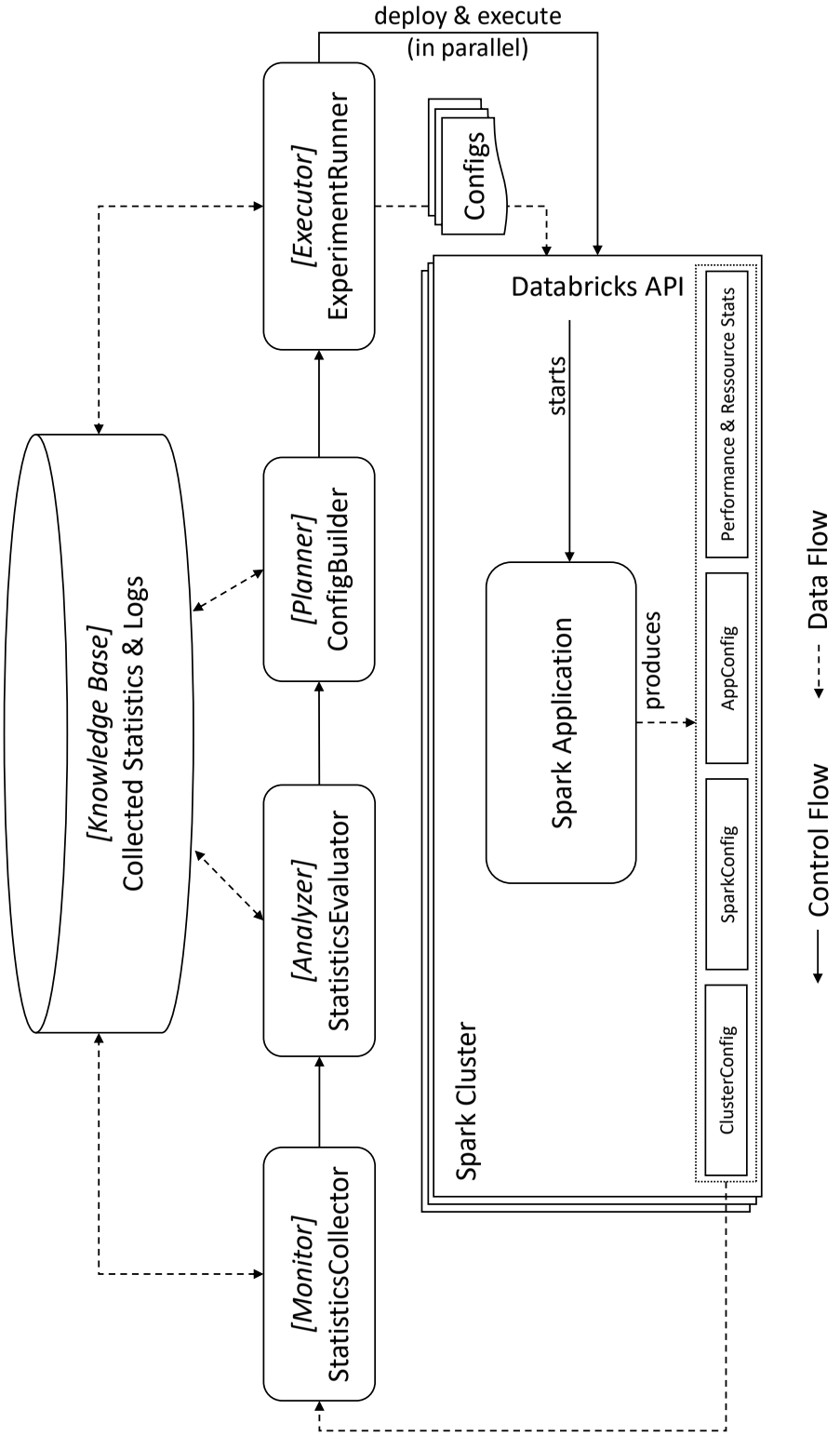

Figure 3.2: Overview about the architecture of the optimization framework implementing the MAPE-K reference model.

The actual Spark application subject to the optimization is then started with the given Spark configuration through the Databricks API. To evaluate the performance of a configuration, it is necessary to collect different statistics during the execution. The collected statistics form the base for the evaluation. Collecting statistics is the purpose of the *StatisticsCollector* component.

Among the collected statistics are the execution time, the Spark logs, resource usage statistics of the driver, and cluster resource utilization during the execution. With the collected data, a more detailed overview of the execution is gained. The collected data is then stored in a data storage including the proposed configuration and the JSON based Databricks API request.

Once all configurations are investigated, the results must be collected and evaluated. The evaluation is realized by the *StatisticsEvaluator* component of the framework. It is responsible for evaluating the configurations' performance and reporting the results to the ConfigBuilder. Since the framework allows any optimization goals in general, evaluating the results depends on the chosen optimization function and must be specified at the beginning. Once the evaluation is finished, the results are reported to the ConfigBuilder. The ConfigBuilder proposes new configuration candidates based on the feedback. Thus, the next iteration starts until a stop criterion is met.

3.4.1 Framework Main Components

This section gives an overview of the most important components of the optimization framework. Their technical realization is described in Chapter 4.

ConfigParser

For the instantiation of the ConfigBuilder, the user-specified parameter space must first be parsed and transformed into the correct format. The instantiation is realized by the *ConfigParser*. It transforms the given, serialized parameter space into the proper format that the ConfigBuilder expects.

The given parameter space must be divided into one or more domains. Each domain must have a `domain_name` key and must contain one or more parameter specifications in the `param_specs` key.

The parameter specification must contain a unique name for the `key` field. Additionally, each parameter specification must have a datatype. The two main datatypes are categorical and numeric datatypes.

In the case of categorical datatypes, the name of the datatype must be set to `categorical_str`. In the `allowed_values` field, a list of all possible categories must be specified.

A numeric parameter specification can have either `float` or `int` as name of the datatype. Optionally, the upper and lower bounds of the value range can be specified. They are necessary to limit the search to a feasible space only. Some Spark parameters need a unit to indicate their byte size. The unit of a numeric parameter can be specified optionally.

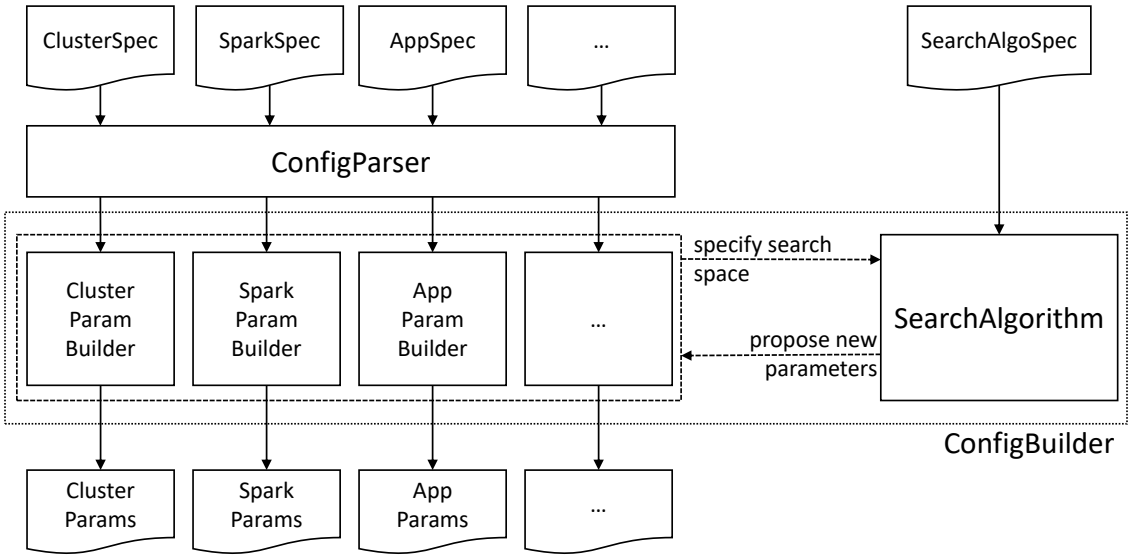

Figure 3.3: Overview of the architecture of the ConfigBuilder component.

Name of datatype	Meaning of datatype
`int`	Integer value
`float`	Floating point value
`categorical_str`	A string of a set of specified categories. Other values than the categories are not allowed.
`bool_str`	Similar to the `categorical_str` datatype but the only allowed candidates are `true` and `false`
`str`	Similar to the categorical_str datatype, but only one candidate is allowed. This datatype is intended for static values that never change.

Table 3.1: Overview of the different datatypes for the parameter specifications.

All existing possible datatypes and their meaning are explained in Table 3.1. Furthermore, the complete domain and parameter specifications are shown in the Appendix A.1.

First, the ConfigParser validates the specified parameters to ensure that the specification is correct. The check is performed for all parameters of all domains. Once the process is finished successfully, the ConfigParser instantiates the ConfigBuilder consisting of the corresponding domain-wise *ParamBuilder* instances (Figure 3.3).

Additional information about the parameter can be specified, like a default value and a boolean `frozen` flag that indicates whether the parameter is mutable. The ConfigBuilder will not alter immutable parameters while searching for an optimized configuration.

A simple, exemplary specification of Spark parameters shows Listing 3.1.

```
{'domain_name': 'spark_config',
    'param_specs': [
        {'default': 1,
         'dtype': {'lower_bound': 1,
                   'name': 'int',
                   'upper_bound': 4},
         'frozen': False,
         'key': 'spark.driver.cores'},
        {'default': 4000,
         'dtype': {'lower_bound': 500,
                   'name': 'int',
                   'unit': 'm',
                   'upper_bound': 4000},
         'frozen': False,
         'key': 'spark.driver.maxResultSize'},
        {'default': 0.6,
         'dtype': {'lower_bound': 0.5,
             'name': 'float',
             'upper_bound': 0.9},
         'frozen': False,
         'key': 'spark.memory.fraction'},
         ...
    ]
}
```

Listing 3.1: Example of a serialized form of Spark parameter specifications for the parameter parsing process.

ConfigBuilder

The ConfigBuilder realizes the Planner component of the MAPE-K reference model. It consists of one or more ParamBuilder components as shown in Figure 3.3. Each ParamBuilder consists of parameter specifications with their respective datatype objects and value ranges.

The ConfigBuilder then serves as the adapter between the ParamBuilder instances and the search algorithm (cf. Figure 3.3). The ConfigBuilder component converts the proposed values into the value formats expected by the ParamBuilder components. Then, the ConfigBuilder component forwards the proposed parameter values by the search algorithm to the corresponding ParamBuilder instances. Next, the ParamBuilder components validate the suggested values and build domain-wise configurations. The domain-wise configurations are then bundled together by the ConfigBuilder and returned. The returned configuration consists of parameter values grouped by each domain. Hence, the validation and search for new parameter values are transparent outside of the ConfigBuilder.

Adapted Genetic Algorithm

The genetic algorithm used to search for improved configuration candidates is an adapted version of the implemented one provided by the `scipy` implementation [VGO+20] based on [SP97]. The reason for choosing a genetic algorithm lies in the characteristics of the optimization problem. Finding an optimized Spark configuration may lead to searching in high dimensional parameter space, depending on how many parameters are considered relevant for the optimization.

Since most of the parameters are continuously valued, there exist an infinite number of parameter combinations that can be selected. An exhaustive search is hence impossible. However, since the Spark cluster and the Spark application are treated as a block box, there is no additional information about how the cluster reacts to the given input configuration apart from empirically observing the Spark cluster. This includes the absence of a gradient indicating the direction of the nearest local minimum. Therefore, a gradient-free algorithm must be selected like the DE algorithm.

However, not all parameters of the Spark configuration are continuously valued. For example, some parameters can only have integer values while others aren't numeric but categorical. This makes the process of optimization even more difficult.

To provide a compatible format for the genetic algorithm, each parameter must be represented as a continuous and bounded numeric variable. This doesn't require any further adaptions for many of the Spark parameters. However, a conversion is required in the case of integer-based variables and categorical variables. For integer values, the conversion from a numeric value to an integer is achieved by simply rounding the proposed numeric value to the nearest integer value.

To convert categorical values into a numeric format, each category is assigned to an integer, like a list index of the position of the current value. To convert the numeric value back to a categorical value, it is rounded to the nearest integer value, which then serves as the index of the categorical value. However, this assumes that the categories belong to an ordinal scale which isn't the case. Therefore, further adaptions, particularly to the mutation strategy, are required.

A different, optional mutation strategy is proposed to handle categorical parameters and is shown in Equation (3.3). Categorical values are of the nominal scale, meaning that they have no order. The only operation between two members of the nominal scale is to compare for equality or inequality. As a consequence, if a categorical parameter is mutated, all other values are equally plausible and probable. Depending on the mutation hyperparameter p, the mutation is performed by drawing a sample from a multinomial distribution. In this distribution, the probability that the current category i remains unchanged is $1 - p$, and the probability of drawing any other category is $\frac{p}{n-1}$ where n denotes the number of available categories of the parameter.

$$p_i = \begin{cases} 1 - p & \text{if } i = i_{prev}, \\ \frac{p}{n-1} & \text{else} \end{cases} \quad \text{with } p \in [0, 1] \tag{3.3}$$

One of the main adaptions of the algorithm is the characteristic of how the algorithm is controlled. The `scipy` implementation takes the optimization function as an input parameter. During the optimization process, it calls the function numerous times per iteration and returns control only after the end of the optimization. Hence, the optimization function is deeply integrated into the algorithm itself.

The deep integration causes problems in the case of using the genetic algorithm to optimize Spark configurations. One of the problems is that with the pure numeric representations of the optimized parameters, they must be converted to the appropriate Spark parameters. Some parameters have additional units; some must be converted to a categorical string value.

Additionally, the proposed configuration must be transformed into a valid JSON format and enriched with additional information to interact with the Databricks API. Using the original implementation, the only way to provide the necessary functionalities would be to integrate them into the optimization function. However, this can lead to tight coupling and complex maintainability in the future. To address this problem, the execution behavior of the algorithm has been changed to handle the particular requirements better. The new execution behavior is illustrated in Figure 3.4.

In the new form of implementation, the genetic algorithm is not responsible anymore for calling the optimization function itself. Instead, after proposing a new set of parameter values, the algorithm's execution is paused until the results of the optimization function have been set externally. The motivation of the adaption is that once the new set of candidates is created, the adapter of the ConfigBuilder component can take over the candidates and transform them into valid Spark parameters.

The transformed set of configuration candidates can then be handed over to the ExperimentRunner component, which converts the set of Spark parameters into a valid JSON format. The conversion is required for the interaction with the Databricks API, which starts to execute the Spark application with the proposed set of Spark parameters. This approach reduces coupling between the components and makes it easier to control what happens with the proposed configuration candidates.

ExperimentRunner

The ExperimentRunner component is responsible for executing the empirical experiments to evaluate the proposed configurations. It realizes the Executor component of the MAPE-K reference model. As input, it accepts the set of proposed configuration candidates. The ExperimentRunner component has two primary purposes. The first is to transform the proposed set of candidates into valid configurations for the target environment. The second purpose is to launch and control the execution of the Spark application with the given configuration.

Since this work uses the Databricks services to deploy Spark clusters with specified configurations, the purpose of the ExperimentRunner is to interact with the Databricks environment. This means transforming the parameters of the proposed configuration candidates to a valid JSON format that fulfills the requirements by the Databricks API. A configuration candidate must be enriched with additional information like the Virtual Machine (VM) types of the driver and the worker nodes, the number of workers, and the Spark version deployed on the cluster.

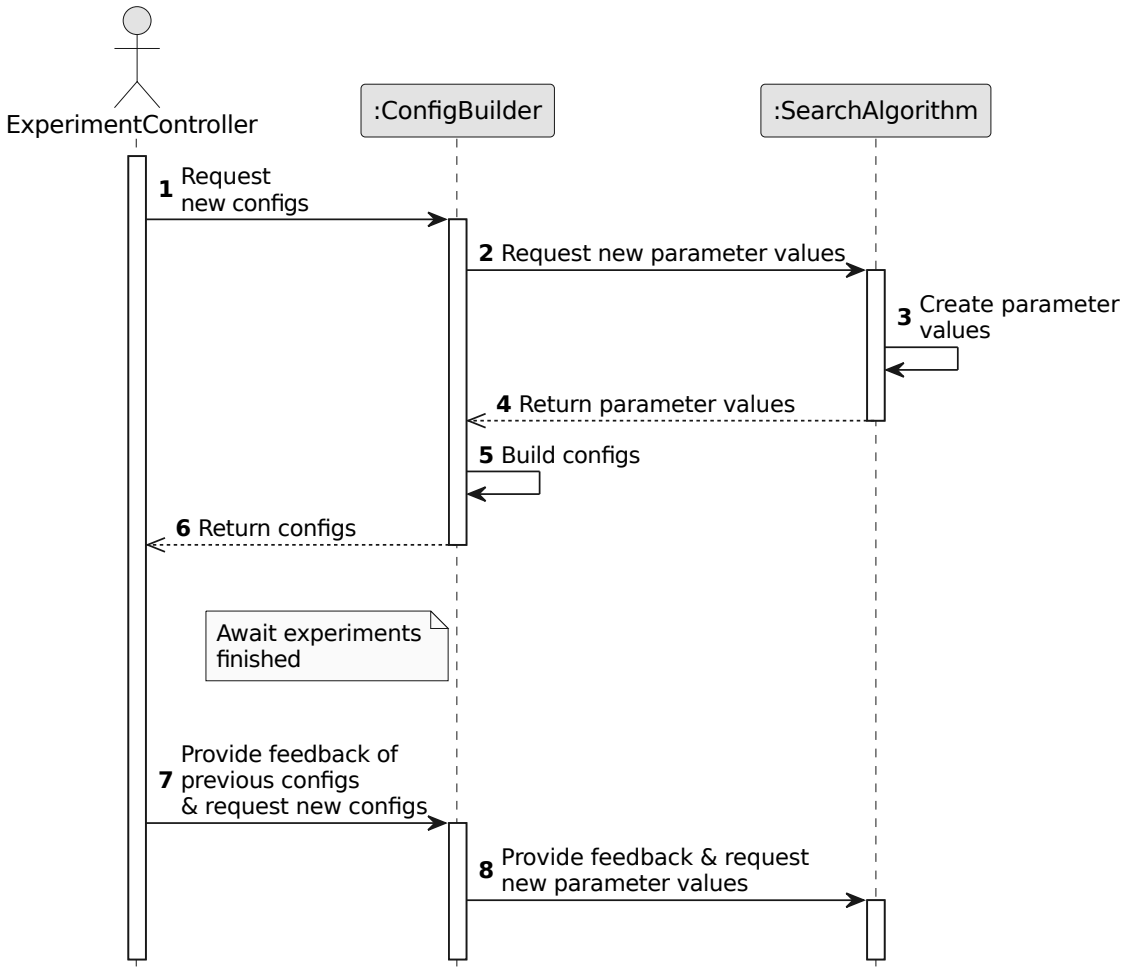

Figure 3.4: Sequence chart about the stepwise search.

Once all additional information is provided, and the JSON object is built, the ExperimentRunner component starts executing the given Spark application using the specified configuration settings. To reduce the optimization time, several configurations can be executed in parallel. The number of possible parallel executions can be controlled with the respective hyperparameter. After launching the experiments, the component monitors the execution of the application and propagates errors to framework user upon occurrence, if specified.

Two types of errors should be distinguished: errors unrelated to the proposed Spark configuration and errors caused by a corrupt Spark configuration. Corrupt Spark configurations may lead to memory errors, extremely long execution times, or cause the cluster to crash. In these cases propagating a timeout or a Spark error to the user isn't necessary. Instead, the configuration shouldn't be considered any further in the optimization process.

This is realized by assigning a very large value as a result of the optimization function. Thus, the optimizer does not consider the specific configuration and will be avoided in the future. This approach's advantage is that errors related to the configuration optimization process won't break the optimization process.

StatisticsCollector

The purpose of the StatisticsCollector component is to collect statistics during the execution of the Spark application. It realizes the Monitor component of the MAPE-K reference model. Thus, the influence of the Spark configuration can be better evaluated by gaining deeper insights into the execution characteristics. The collected statistics can later be used to evaluate the configuration to provide feedback to the ConfigBuilder but can also serve for further offline analysis. To the collected statistics belong

- the total execution time,
- the Spark logs from the driver and the workers,
- the statistics that can be accessed via the Spark API like statistics about the executors, the jobs, and the stages of a Spark application,
- a heartbeat log to reconstruct at what time the execution of the Spark application was aborted regarding the total execution time.

StatisticsEvaluator

The "StatisticsEvaluator" component is responsible to evaluate all collected statistics. It realizes the Analyzer component of the MAPE-K reference model. The type of statistics that need to be evaluated highly depends on the chosen optimization goal.

It is the function that traverses through all the experiments to collect and parse the required data. Such data can be the total execution time of the Spark application, the time of single stages or jobs, or the number of shuffled bytes. The other primary purpose of the StatisticsEvaluator is to aggregate the results. The aggregation combines the results of several repetitions into a single value. Popular aggregation functions are the mean or median aggregation.

Knowledge Base

For the realization of the knowledge base (Figure 3.2), no special component is required in the Databricks runtime environment since shared storage is already available. Every component of the framework already has access to it. The collected statistics are stored in the storage and can be evaluated by the StatisticsEvaluator component.

4 Prototypical Implementation of the Framework

The chosen language for the implementation is Python, a flexible language that covers many different requirements of this work by offering powerful libraries. Examples of these requirements are performing network communication and interacting with REST APIs, data processing and statistics, and scientific computing.

Statistical data processing is supported by the commonly known `pandas` and `numpy` libraries. For the adapted genetic algorithm the implemented version provided by the `scipy` module [VGO+20] is used. High-level network communication like interacting with REST APIs is supported by the `requests` module. Finally, Python also supports Spark via the `pyspark` API which makes interacting with Spark in Python convenient.

The implementation of the components described in Section 3.4 are illustrated in the Unified Modeling Language (UML) class diagrams shown in Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5 where also the realization of the MAPE-K components are shown.

Implementing the framework as generically as possible is an important aspect of the implementation. For this reason, the implementation uses interfaces (protocol classes in Python) and abstract base classes. This reduces the coupling between the components and standardizes data formats needed for interaction. Therefore, future framework extensions are possible as described in Section 6.3.

4.1 Implementation of Orchestration

The core component that orchestrates the optimization logic is the ExperimentController class. Once the components have been instantiated, it coordinates the functionalities of the components shown in the activity chart in Figure 3.1. After the instantiation it will request new configuration candidates from the ConfigBuilder (Figure 4.3) calling the `get_next_configs()` method. It forwards the configuration candidates to an instance of the DatabricksExperimentRunner class and starts the execution of the experiments by calling the `run_experiments()` method.

Once the DatabricksExperimentRunner instance reports the termination of the experiments, the ExperimentController instance starts the evaluation of the experiments by calling the `evaluate()` method of the FileBasedStatisticsEvalutor instance. The ExperimentController instance forwards the results once again to the ConfigBuilder instance, calling the `get_next_configs()` method. This marks the beginning of a new iteration.

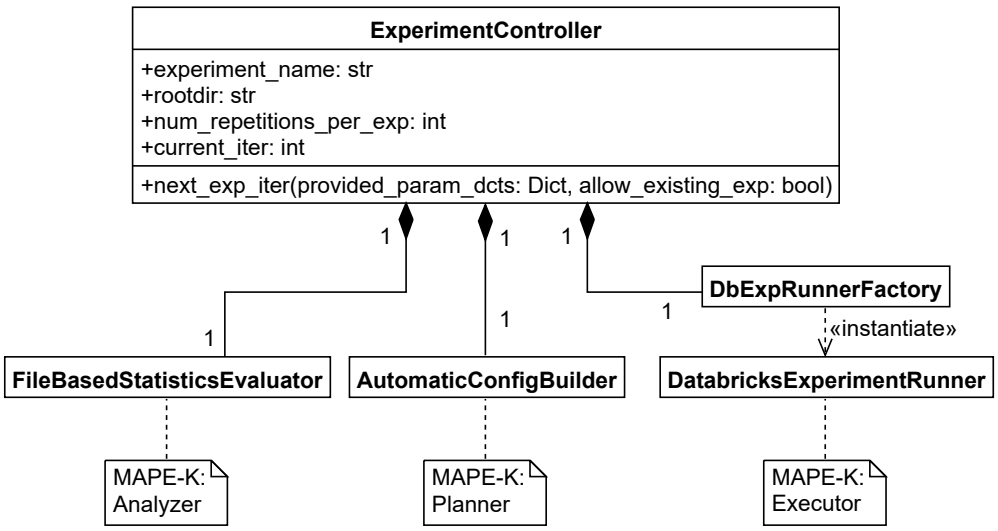

Figure 4.1: UML class diagram showing the relationship between the core classes of the optimization framework.

4.2 Implementation of MAPE-K Components

The *Monitor* component is implemented by the StatisticsCollector class, shown in Figure 4.5. It implements an asynchronous processing model for collecting statistics and observation data. The reason for using asynchronous execution is to use as few resources as possible to have little impact on the execution of the actual Spark application that is monitored. It consists of one or more collectors representing a source of statistical data queried after certain time intervals. To avoid writing small and numerous data to disk at a high frequency, the CachedStatsCllctr class is introduced. It caches the collected data until a threshold is reached. The amount of written data is hence increased, and the frequency of the writes decreased. This makes the process of collecting statistics more efficient.

The *Analyzer* component is implemented by the FileBasedStatisticsEvaluator class, as illustrated in Figure 4.2. Its purpose is to evaluate the collected statistics. The term *FileBased* is used because this class evaluates the file-based statistics of the shared storage.

The FileBasedStatisticsEvaluator relies on the DataGetter interface (Figure 4.4) as data source. The DataGetter interface is responsible for providing the evaluation function of the experiment. It returns a single numerical value subject to the optimization process and makes different configurations comparable.

Using an interface for the evaluation makes the framework very flexible since providing a new, custom-defined objective function is convenient. Exemplary realizations of the DataGetter interface are evaluation functions that retrieve the execution time or the number of shuffled bytes during execution.

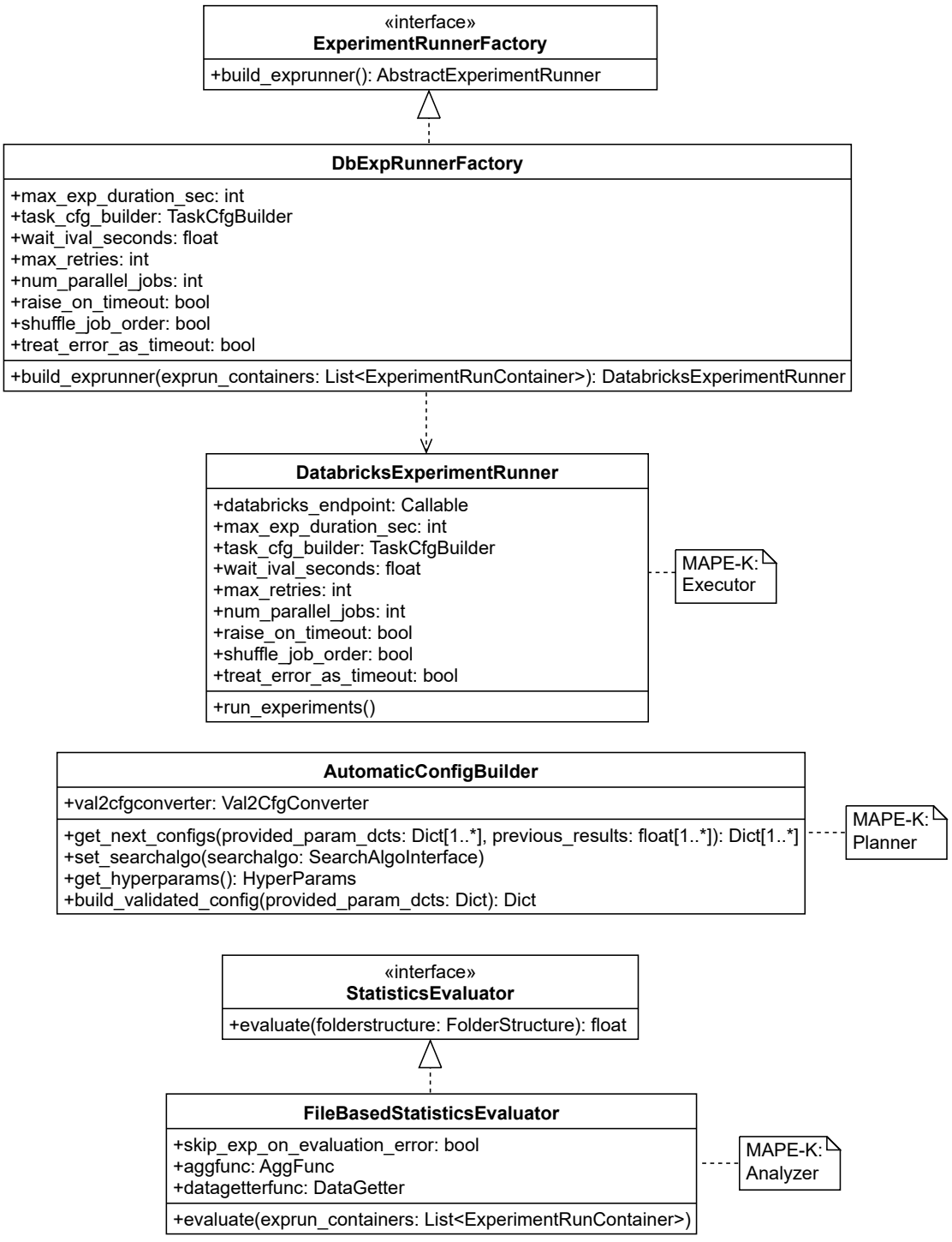

Figure 4.2: UML class diagram showing the implementation of the core classes for the realization of the chosen architecture.

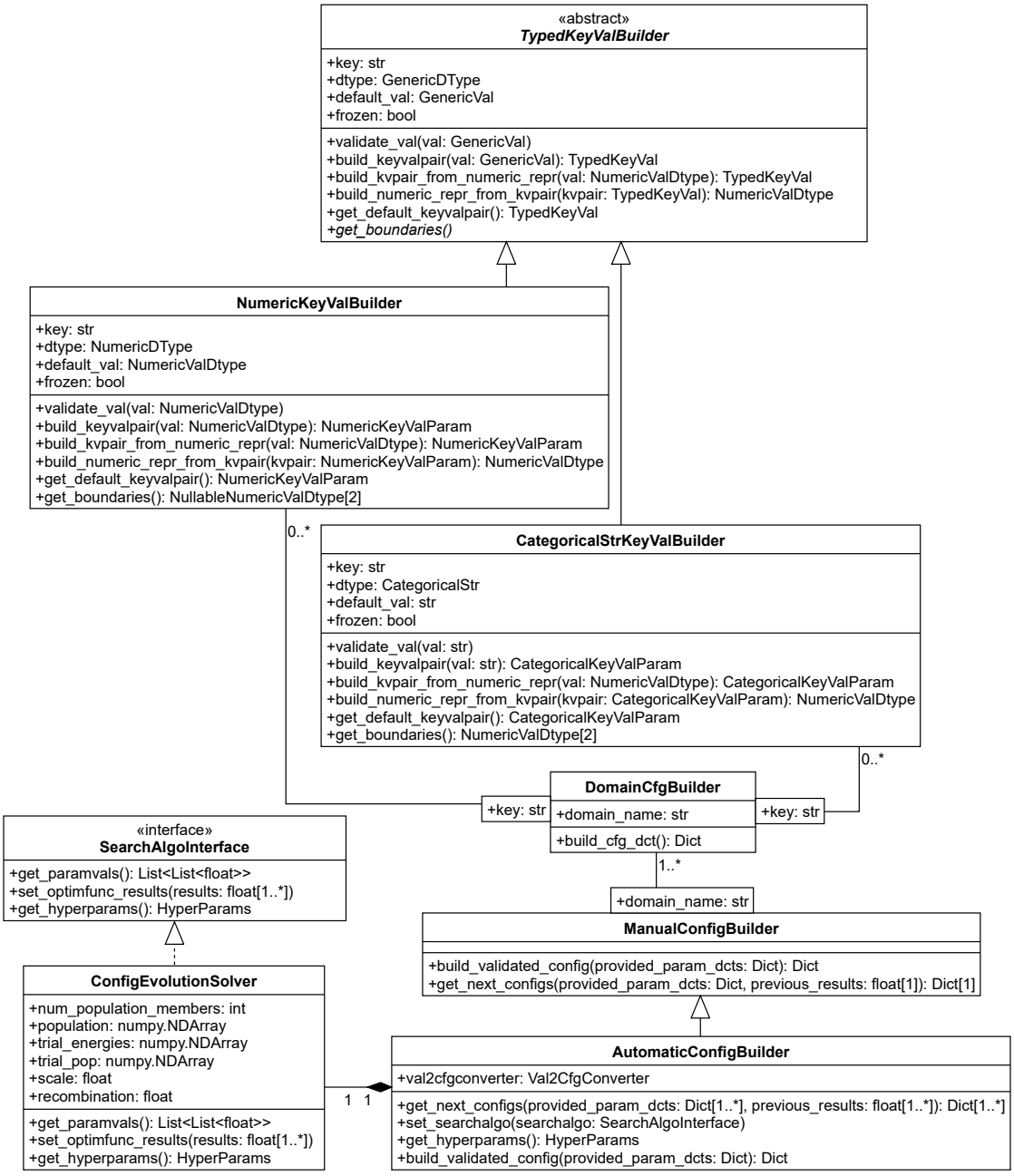

Figure 4.3: UML class diagram of the ConfigBuilder class with its most important utility classes.

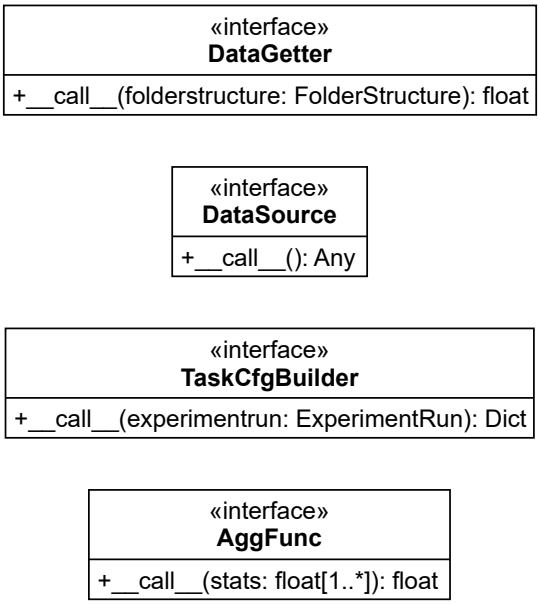

Figure 4.4: UML class diagram of important interfaces.

The evaluated data is then aggregated using the AggFunc interface. This function aggregates multiple repetitions of an experiment to a single value. Due to the use of an interface, the aggregation logic can be conveniently adapted. The ExperimentController instance then forwards the aggregated statistics to the ConfigBuilder, which suggests new configuration candidates.

The ConfigBuilder has two implementations; the *AutomaticConfigBuilder* class and the *ManualConfigBuilder* class.

The key difference between the two classes is that the AutomaticConfigBuilder class is designed to search autonomously for new configuration candidates. In contrast, the ManualConfigBuilder class is designed to use an externally provided set of parameter values instead of a proposed one (e.g., by a search algorithm). It then enriches them with additional default values if not provided.

The possibility of manually specifying configurations is useful when specific, user-defined configurations should be investigated. In this case, the ManualConfigBuilder class will enrich the configuration with additional default values if no values for the corresponding parameters are specified. Both classes validate the given configuration values to assert the parameter values lie within the specified parameter space.

The AutomaticConfigBuilder class implements the *Planner* component.

By calling the `get_next_configs()` method and providing the evaluated and aggregated statistics the AutomaticConfigBuilder is triggered to propose new configuration candidates. Due to the SearchAlgoInterface, the AutomaticConfigBuilder can request new configuration candidates from the search algorithm without knowing anything about the algorithm. It provides the evaluated statistics to the algorithm by calling the `set_optimfunc_results()` method of the interface. It then requests new configuration candidates by calling the `get_paramvals()` method.

The returned numeric parameter values are then converted into domain-wise configurations using the Val2CfgConverter class. The proposed parameters are then forwarded to the DomainCfgBuilder instances. The DomainCfgBuilder instances validate and enrich the provided parameter sets. Once this is completed, the final configuration can be returned to the ExperimentController.

The DatabricksExperimentRunner class implements the *Executor* component. Since this work uses a Databricks environment, the component that receives the configuration candidates is the DbExpRunnerFactory. The DbExpRunnerFactory component will then construct a DatabricksExperimentRunner instance which is responsible to convert the given configuration into a valid JSON format for the interaction with the Databricks API (see Figure 3.2).

Calling the TaskCfgBuilder interface builds a part of the required JSON for the interaction with the Databricks REST API. The TaskCfgBuilder returns a JSON object for a new "notebook task". The interface allows specifying the actual notebook task, including which notebook should be executed and which arguments should be used. This makes it convenient to quickly set up the framework for optimizing different Databricks notebooks containing different Spark applications.

The technical realization using Databricks notebooks is described in detail in Section 4.3.

The resulting JSON object is then passed to the Databricks API to start the execution of the Spark application with the given configuration. Each experiment then represents a Databricks-Job which gets the JSON as input containing the Spark configuration, among other things.

The implementation of the *Knowledge Base* is partially covered by the FolderStructure class. Although it doesn't abstract completely away the technical details, it abstracts away the concrete paths of files that are created during the optimization process.

4.3 Technical Realization

The concrete technologies used to realize the optimization framework are illustrated in Figure 4.6. The whole framework is located in a private Azure cloud instance. The cloud instance provides, among other things, the VMs that are used to host the Spark driver and workers onto. A "DataLake Storage Gen2" is used to read and write persistent data. The advantage of this storage system is that within the cloud instance, it is available in the Databricks environment as a mounted drive and is hence accessible like any other local storage. This enables all components to write and access persistent data, and the DataLake serves hence as the *KnowledgeBase* like in the MAPE-K reference model, shown in Figure 2.3.

For the process of optimizing Spark configurations, two Databricks notebooks are used. The main part of the optimization framework is located in the first notebook - the "optimization notebook". The notebook contains all instances belonging to the ExperimentController. In terms of the MAPE-K reference model, this means that the realizations of the Analyzer, Planner and Executor reside in this notebook.

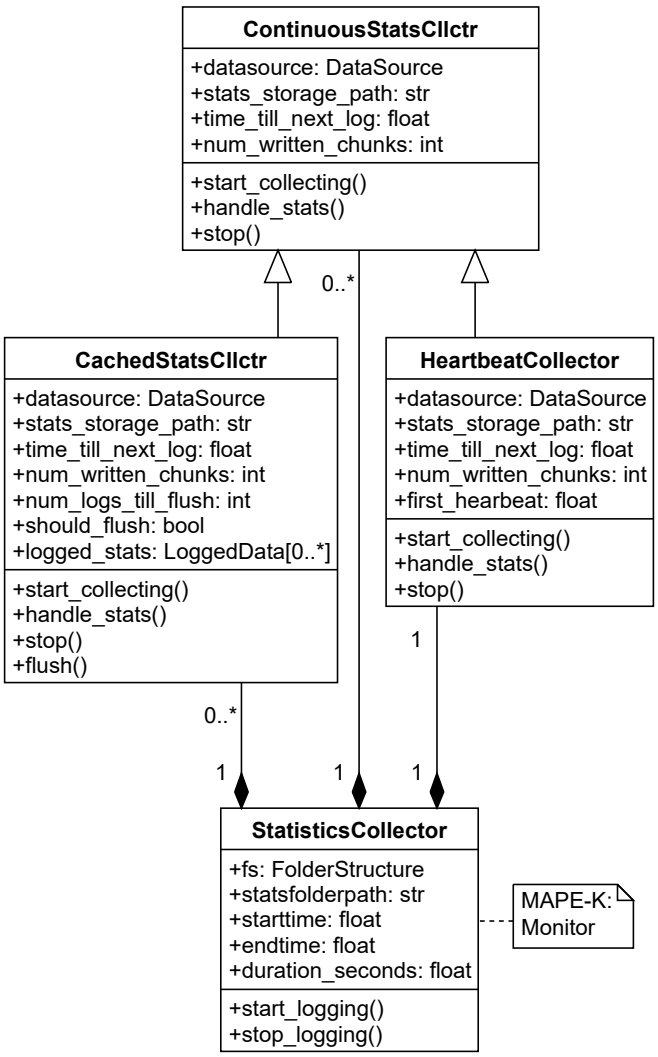

Figure 4.5: UML class diagram about the implementation of the StatsCollector class.

Once the optimization process is started, the optimization framework requests a newly instantiated Spark cluster via the Databricks API. This is realized by calling the *jobs/runs/submit* endpoint of the API [Dat22b] and passing the proposed Spark configuration to the API. This call will then start to execute the previously specified second notebook - the "Application Notebook".

This notebook contains the actual Spark application that is subject to the optimization and an instance of the StatisticsCollector class. The StatisticsCollector class is a wrapper around the actual Spark application. The process of logging is started before the execution of the Spark application. After the termination of the Spark application, logging and collecting statistics is stopped, and the collected data is stored in the DataLake Storage.

Once the StatisticsCollector has finished storing the collected data, the run of an empirical experiment is finished. Via the Databricks API, the termination is visible to the ExperimentRunner of the optimization framework. For the continuation, two possibilities exist. The first one is that the

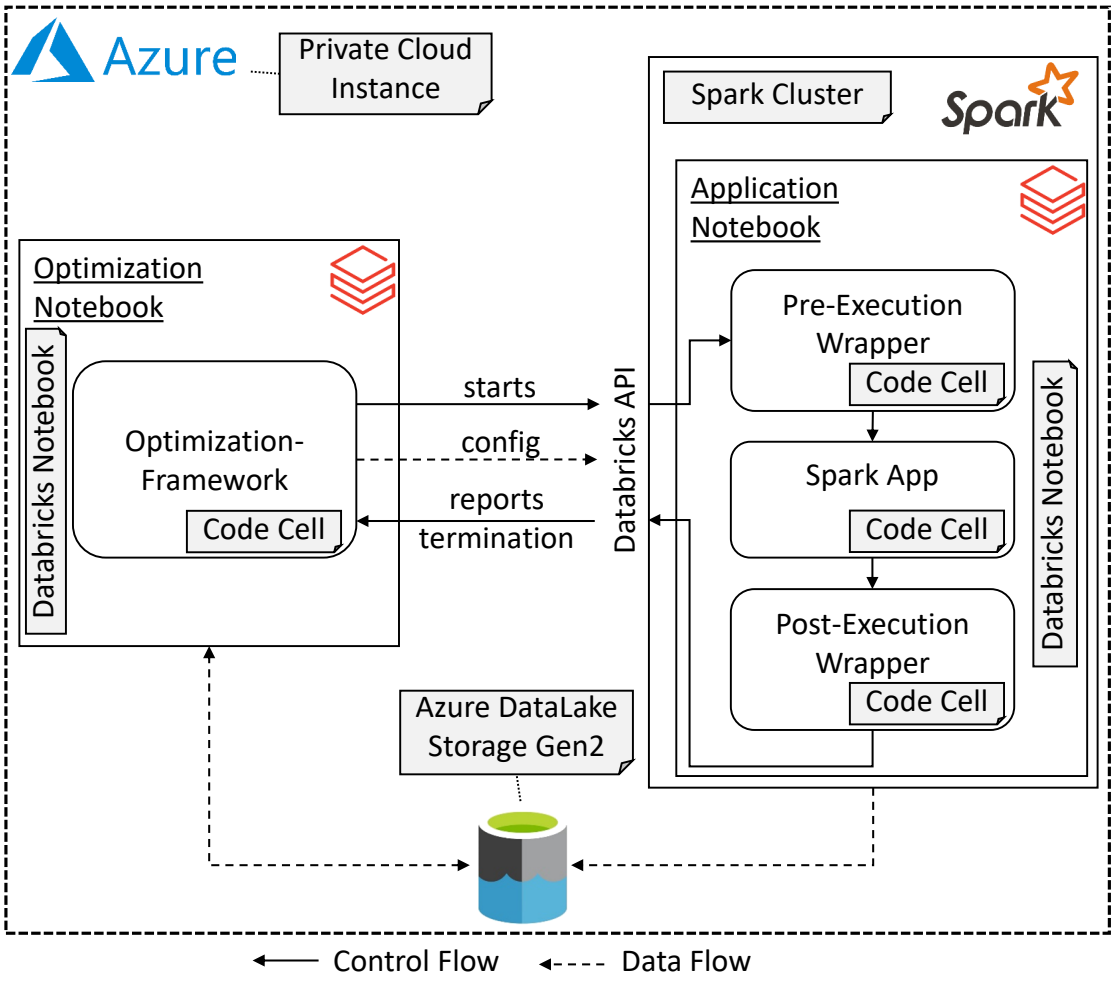

Figure 4.6: Used technologies for the technical realization of the optimization framework (Dabricks icon: [Agr22], Azure icon: [Cor22], Spark icon: [Fou22b], DataLake icon [mag22]).

ExperimentRunner still has other runs in its queue of experiments that aren't executed yet. Then the described execution loop is started again with the new run. The second one is that all runs of all configuration candidates have been completed. Then the StatisticsEvaluator starts to read out the relevant information from the DataLake storage. The evaluated statistics are then passed to the AutomaticConfigBuilder to propose a new set of configurations. The execution loop shown in Figure 4.6 is then started with the new configuration candidates.

The reason for splitting the framework into two notebooks is necessary to keep the state of the framework. The second notebook shown in Figure 4.6 is deployed and executed on a newly instantiated Spark cluster. The StatisticsCollector instance needs to be physically close to the Spark application to gather the relevant statistics. It must be deployed together with the Spark application on the same cluster. This requirement contradicts the one of keeping the state of the framework to handle the execution of multiple Spark clusters and perform the optimization itself. Splitting

the framework solves this problem since the components that need to keep their state, such as the AutomatConfigBuilder instance, are placed in the optimization notebook, which keeps its state during the optimization process.

The wrapper-like behavior of the StatisticsCollector in the Application Notebook is achieved using different code cells in the notebook. The cells are executed sequentially, as shown in Figure 4.6. A useful characteristic of Databricks notebook is the independence of concrete programming languages. Databricks notebooks allow mixing different programming languages within a single notebook by placing the contents of the different languages in separate cells.

This means that the choice of Python as the language for the implementation doesn't restrict the framework to optimize only `pyspark` applications. Instead, every Spark application based on any of the supported programming languages by Spark can be subject to optimization.

Another advantage of using a Databricks notebook for the Spark application is that they conveniently allow specifying further externally passed-in arguments for the execution, which are accessible within the code cells. Thus, additional arguments can be passed to the Spark application. They can be static or also subject to optimization. The optimization framework already supports the ability to specify different domains for the optimization, as shown in Figure 3.3. The additional parameters can be passed to the Spark application via the Databricks API.

Theoretically, the first notebook is not necessary as the design of the framework doesn't require it. Any runtime environment that provides access to the shared storage and the Databricks API can be potentially used. However, due to security restrictions of this work, a Databricks notebook as a runtime environment was required.

5 Case Studies

This chapter presents the results of applying the framework in a case study to demonstrate the practicability of the framework. Two exemplary use cases are selected for this purpose.

5.1 Validation of Framework

Several validation experiments have been performed to prove that the framework, particularly the DE algorithm, can optimize a configuration for a given optimization. They can be separated into two categories: applying the framework to artificial test functions and applying it to a real Spark optimization problem. The goal of the validation is to show that the optimization framework can find an improved configuration over the initial configuration with the implementation described in Chapter 3. Ideally, this would be the global minimum of the test functions. The settings, shown in Table 5.1, are applied the DE algorithm. The search interval for each parameter is chosen as $[-5, 5]$.

One of the artificial functions that was selected to validate the implemented framework is the "Rosenbrock" function [Ros60]. The Rosenbrock function, illustrated in Equation (5.1), is a function that maps an arbitrary dimensional vector to a scalar $f : \mathbb{R}^d \mapsto \mathbb{R}$ where d denotes the number of dimensions of the definition space and $d \geq 2$.

$$f(x_1, x_2, ..., x_d) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \tag{5.1}$$

The global minimum of the Rosenbrock function lies at $\mathbf{x}_{min} = \begin{bmatrix} 1 & 1 & ... & 1 \end{bmatrix}$ with $\mathbf{x}_{min} \in \mathbb{R}^d$ and $f_{Rosenbrock}(\mathbf{x}_{min}) = 0$.

The second artificial function that is used for the validation is the "Styblinski-Tang" function. It is defined in Equation (5.2). Like the Rosenbrock function, the Stylinski-Tang function maps an arbitrary dimensional vector to a scalar $f : \mathbb{R}^d \mapsto \mathbb{R}$ where d denotes the number of dimensions of the definition space.

$$f(x_1, x_2, ..., x_d) = \frac{1}{2} \sum_{i=1}^{d} x_i^4 - 16x_i^2 + 5x_i \tag{5.2}$$

Hyperparameter	Value
Mutation	Randomly drawn from uniform distribution $U(0.5, 1)$ every iteration
Recombination	0.7
Population Size	5
Iterations	300

Table 5.1: Overview over the different chosen hyperparameters for validating the DE algorithm.

The global minimum of the Styblinski-Tang function is located at $\mathbf{x}_{min} = \left[x_{min} \dots x_{min}\right]$ with $x_{min} = -2.903534$ and $\mathbf{x}_{min} \in \mathbb{R}^d$.

The achieved results are illustrated in Section 5.1.

In the case of the Rosenbrock function, the figure shows the absolute distance between the result achieved by the current configuration to the global minimum. In the case of the Styblinski-Tang function the results have been normalized by applying the transformation $y_{normalized} = \left|\frac{f(\mathbf{x}) - y_{min}}{y_{min}}\right|$. Thus, the transformed values show the distance of the optimized configurations to the global minimum relative to the global minimum. This makes the results also for different amounts of dimensions comparable. The results show that for both artificial functions, the differential algorithm was able to improve the configuration significantly. In the case of 5 and fewer dimensions, the global minimum for both test functions is found. For the case of higher dimensions, the global minimum is not reached anymore. However, the differential algorithm with the chosen settings can still significantly improve the initial configuration.

To further investigate the framework's capabilities and validate that the framework can optimize real Spark configurations, the framework is tested using a real dataset. In this dataset, several configurations were tested. The optimization goal was to reduce the number of shuffled bytes in the cluster. This selection has been made because this optimization goal is stable and time-independent (see Section 5.3.4 for the motivation). The dataset simulates the real Spark cluster to avoid costly experiments. Due to the robustness of the chosen optimization goal (Figure 5.6), the simulation will produce the same results in comparison to a real cluster.

A simple multidimensional regression model is then trained on the dataset and serves as an oracle to predict the shuffled bytes in the cluster. This is necessary as not for every combination of parameter values has an experiment been performed. Hence the model can fill the gaps where no real data is available for the chosen combination of values. To further reduce the gaps in the dataset, the framework is configured only to choose values for each parameter contained in the dataset. This is achieved by selecting categorical values (see Table 3.1) for each parameter, where each category represents the value in the dataset for the respective parameter. This procedure is applied to two different datasets.

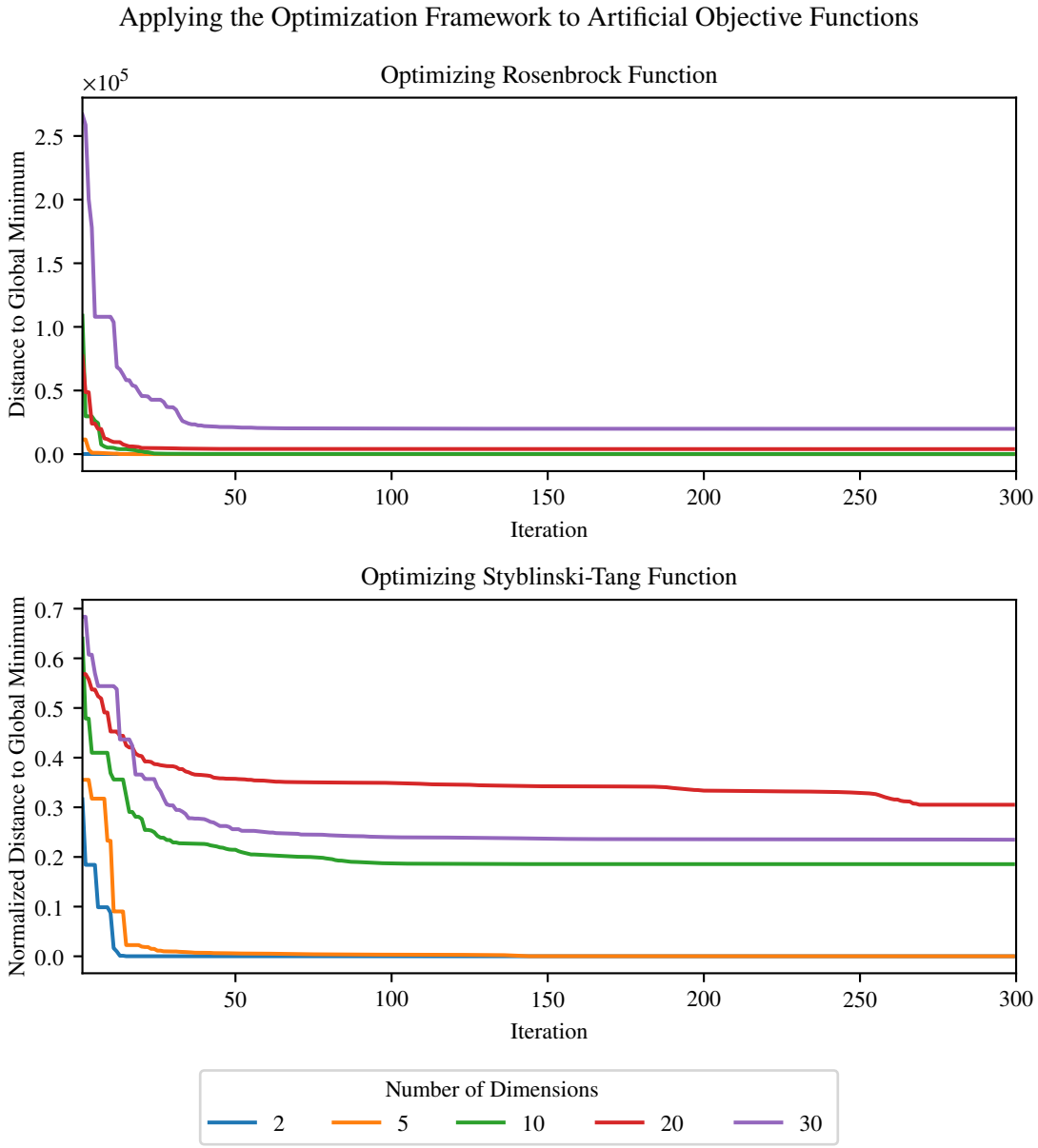

Figure 5.1: Achieved results when the optimization framework is applied to artificial functions to find the configuration with the lowest resulting function value.

Five parameters are used for the validation, namely

- `spark.io.compression.codec`,
- `spark.io.compression.lz4.blockSize`,
- `spark.shuffle.compress`,
- `spark.io.compression.snappy.blockSize` and
- `spark.shuffle.file.buffer`.

The validation results using a real dataset are shown in Figure 5.2.

The graphs show the process of optimization for different starting positions. In the case of the first dataset, all configurations reach the global minimum. For the second dataset, similar effects can be observed. Here the framework can optimize the initial configurations until almost the global optimum is reached.

The validation results show that the framework can optimize configurations of artificial objective functions and real Spark application data. The validation of the framework is hence successful. However, a trend can be observed when the framework is used to optimize artificial test functions. Optimization is more difficult for a higher number of dimensions, and the global minimum is not reached anymore. Despite this effect, the configurations in all scenarios could be improved. This means that the number of dimensions should be kept as low as possible for future optimizations to achieve the best optimization results. The second important conclusion is that if the potential for optimization exists, the framework can find at least an optimized configuration.

5.2 Concept of Experiments

Some general conceptual decisions that regard all performed experiments are discussed in this section.

Virtualization plays an important role in a cloud environment to offer every customer the same quality of service. Virtualization is an abstraction of the physical hardware to decouple physical hardware from the applications and programs executed on it. Thus, every user in the cloud can use the same machine and hardware without interference from other customers. One of the consequences of virtualization may be that the physical hardware is shared.

Sharing physical hardware can lead to significant consequences for all guests that use the provided hardware in the form of VMs. Demanding applications can slow down other guest applications. However, one crucial requirement for optimizing Spark configurations is the results' reliability. This means that results must be reproducible. Since the Spark cluster resides on VMs, potentially varying execution times are very likely and expected. Additional variances may occur because the network is also shared between several cloud customers. Network traffic does not only occur during the shuffling of data but also when data is written to disk. This variance exists because the storage is connected via a network with the Spark cluster. This means lower throughput in the network will

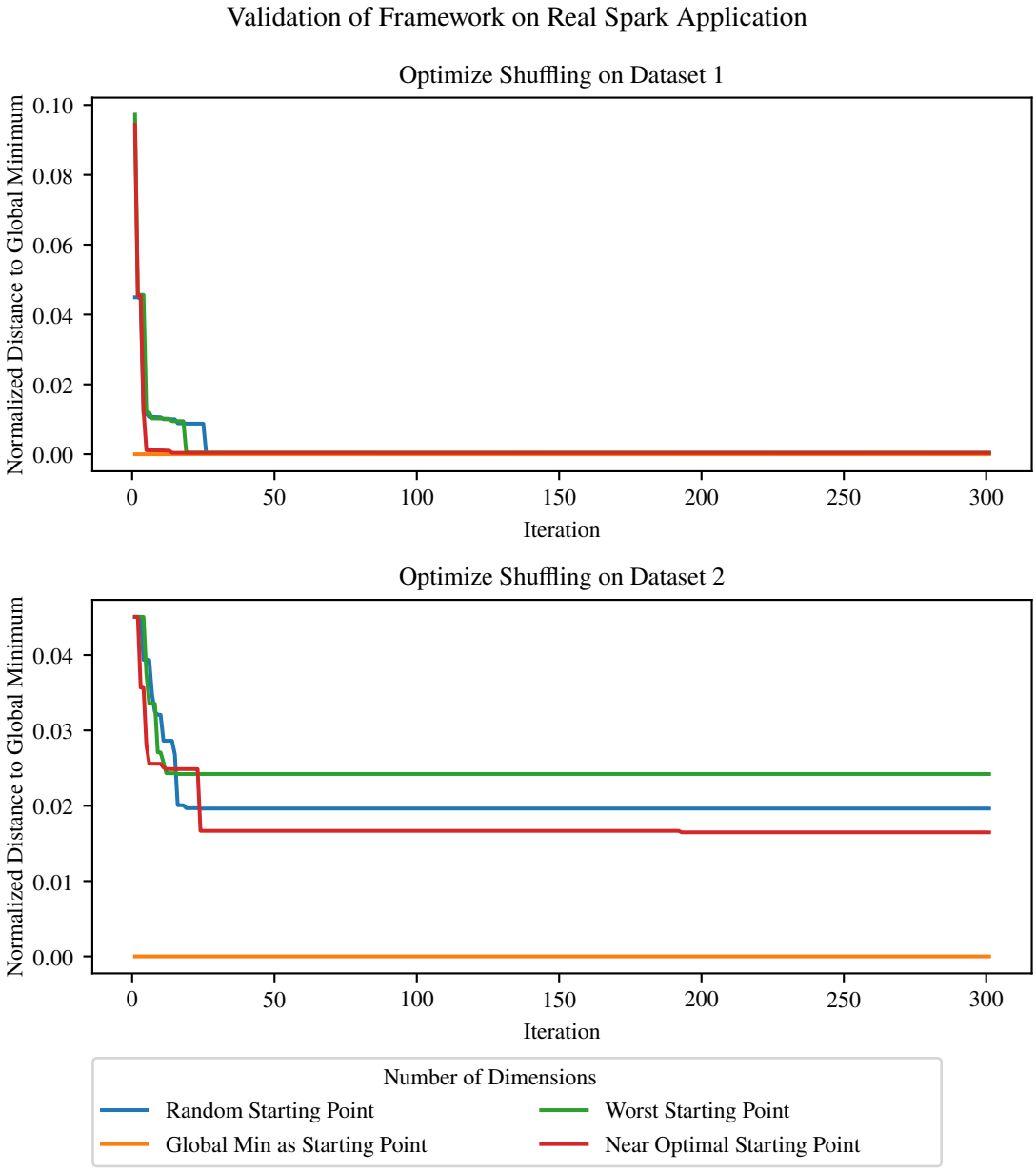

Figure 5.2: Achieved results when the optimization framework is applied to a real Spark application to find the configuration with the lowest resulting cluster shuffling.

also impact the execution time of the cluster. Therefore, an investigation of the runtime variances with fixed input data and using the standard Spark configuration is necessary. It will be executed for each case of the case study.

A single trial of a new configuration candidate is not enough to evaluate its performance. It cannot be distinguished between whether the configuration itself is the reason for the resulting execution time impact or external factors are the cause that is unrelated to the configuration. To encounter this problem, several trials of each configuration candidate are performed. The intuition of trying out a configuration candidate several times is that the execution times follow a type of distribution. Aggregating the execution times by determining the mean or average value of the execution times will then lead to a reliable execution time. With this execution time, it is possible to evaluate each configuration candidate. The concrete number of repetitions in this work is set to 10.

Due to the characteristics of a heterogeneous cloud environment, the time until a Spark cluster is set up and usable may vary highly since the virtualized nodes have to be acquired first. Therefore the start of the execution time is the point of time when the Spark cluster is completely set up and usable, and all additional libraries and packages are fully installed.

The Databricks runtime version used in the case studies is `10.4 LTS` which includes `Spark 3.2.1.` and `Scala 2.12`.

To ensure that there are no influences between the experiments and their repetitions, each repetition is executed on a new Spark cluster. Thus, it is ensured that no caches are already filled, or other remainings of past executions still exist and influence the results of the experiments.

For the following case studies, some real production scenarios where Spark is used have been selected. They resemble an example of Spark applications that are used in production. While standard benchmarks may be used for comparison and to prove the success of a concept, optimizing Spark applications in a production environment have a higher relevance for the applicants of Spark. Optimizing such applications has challenges as each application may differ significantly from the others. Therefore, using such real-world examples is highly relevant and interesting for optimizing Spark applications used in production.

5.3 Optimization of Binary Data Processing

One of the exemplary use cases in which the optimization framework was applied is the conversion of time series measurement data from the proprietary MDF format (described in Section 2.5) into a relational format, namely the Optimized Row Columnar (ORC) format. The relational format serves then as a base format for future analysis. The reason for this conversion is that Big Data frameworks like Spark are well optimized on this format in contrast to proprietary formats, and the converted files can then be used for further analytics.

Name of VM Type	No. of Instances	No. of Cores	Memory Size [GB]	Purpose
StandardF4	1	4	4	Spark Driver
StandardF8	2	8	8	Spark Workers

Table 5.2: Overview over the used VM types.

Hyperparameter	**Value**
Mutation	Randomly drawn from uniform distribution $U(0.5, 1)$ every iteration
Recombination	0.7
Population Size	5
Iterations	30
Number of Repetitions	10

Table 5.3: Overview of the different chosen hyperparameters of the experiment.

Due to a large amount of measurement data, the costs of converting them from the proprietary MDF format into a relational big data format depend mainly on the time that is needed for the conversion. Every improvement that shortens the runtime of the conversion will lead to a reduction in costs. Due to a large amount of data, even minor improvements that shorten the runtime can significantly impact the total costs.

5.3.1 Design of Experiments

In this section, the setup of the performed experiments is described. All the experiments were executed in a private cloud environment, based on the "Microsoft Azure Cloud". In this cloud, among other things, file storage is offered and computing nodes on demand in the form of VMs. On top of this service offerings, the Databricks runtime environment is offered as described in Section 2.2. For the performed experiments of this use case, the VM with the smallest available resources was chosen to host the Spark driver, and the second smallest VM type was chosen to host the worker. In total, two worker nodes were chosen for the Spark cluster. The reason for this choice was that the price of the usage of the VMs depends on the runtime itself and the type of resources selected for the VM. VMs with many cores and large memory can become rather costly, particularly if many clusters are deployed during the optimization process. However, the selected VM-type is cheap and therefore is appropriate for an empirical optimization approach where the Spark application has to be executed many times. The actual chosen VM types are shown in Table 5.2. The underlying physical machines and the pricing of the cluster types can be found in [Mic22].

The configuration settings of the DE optimization algorithm are illustrated in Table 5.3 and the Spark configuration parameters are shown in Table 5.4. The median value was chosen as the statistical aggregation function. This means that of the ten repetitions, the median is selected as the single aggregated value representing the configuration's performance.

parameter	lower_bound	upper_bound	dtype	unit	optimized_val
spark.driver.cores	1	4	int		1
spark.driver.maxResultSize	500	4000	int	m	3631m
spark.driver.memory	1000	2923	int	m	1466m
spark.driver.memoryOverhead	0.1	0.5	float		384
spark.executor.cores	1	8	int		7
spark.executor.memory	1000	8874	int	m	5211m
spark.executor.memoryOverhead	0.1	0.5	float	m	733m
spark.files.maxPartitionBytes	1000000	512000000	int		96101579
spark.memory.fraction	0.5	0.9	float		0.84
spark.memory.storageFraction	0.5	0.9	float		0.59
spark.reducer.maxSizeInFlight	1	500	int	m	84m
spark.shuffle.file.buffer	1	500	int	k	83k
spark.network.maxRemoteBlockSizeFetchToMem	200	512	int	m	272m
spark.broadcast.blockSize	1	500	int	m	23m
spark.default.parallelism	2	16	int		16
spark.storage.memoryMapThreshold	1	128	int	m	3m
spark.rpc.message.maxSize	1	500	int		127
spark.io.compression.lz4.blockSize	2	256	int	k	32k

Table 5.4: Used Spark parameters with their specified value range for optimizing binary data processing execution time, showing the parameter values with the lowest shuffling traffic.

5.3.2 Reproducibility

Several measurement files of different input sizes are used to investigate the variances regarding the execution time of binary data conversions. The motivation is to evaluate the file size's effect on the execution times variances. If the variances prove to be independent of the file size or execution time, then longer execution times and larger files would be more robust against variances, meaning that the results would be more reliable. This would reduce the effort to analyze noisy execution times.

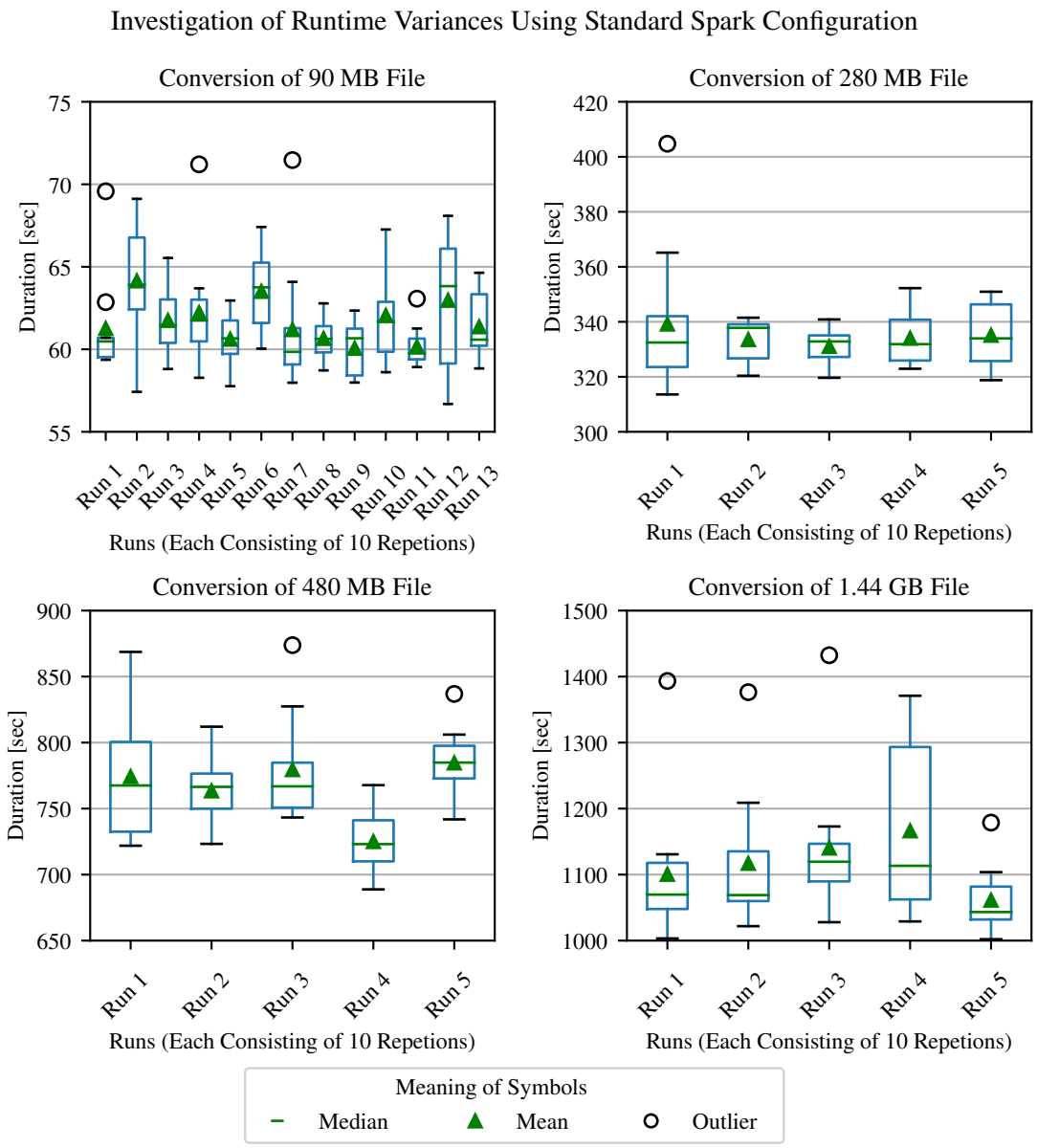

Figure 5.3: Overview over the execution time variances of various input data sizes and fixed standard Spark configuration.

The variances of the performed investigations are illustrated in Figure 5.3 and Figure 5.8. Each diagram represents one file of a different size that was converted. Each boxplot in the diagram represents the distribution of one run where the file was converted 10 times and then the execution times were collected.

For the conversion of the smallest file, it can be denoted that while many runs have similar execution times, some runs tend to have broad distributions, especially in the direction of longer execution times. For some runs, extreme outliers can be observed, meaning that one execution time was significantly longer than the rest. Using the mean value to aggregate the different execution times of the first run into a single value can be misleading as the mean value is typically very sensible towards outliers. In this case, the mean value lies even outside the distribution. A more robust aggregation method is using the median value as it is not sensible towards outliers. However, even when using the median as the aggregation function, the diagram shows that there are more than 4 seconds of difference between run seven and run twelve, which is around 6% difference.

One reason for the previously mentioned runtime deviations might be the relatively short execution time of the Spark application. In this case, small variances are dominant compared to the actual execution time, making it challenging to find the accurate execution time that would result if no variances exist.

One approach to this phenomenon is using a longer-running Spark application. This is why several larger files were tested in the conversion too. For the conversion of a 280 MB measurement file, the occurrence of an outlier with a remarkably long execution time can be observed. In this case, the outlier has roughly a 25% higher execution time than the first run's median. For this file, the runs' median values are distributed relatively similarly, around 330 seconds.

The results of converting the 480 MB file are unexpected. Apart from two outliers in run three and run seven, the distributions of run three and four deviate significantly from each other. The execution times of the fourth run were significantly shorter than those of all other runs.

The execution times for converting the largest measurement file are illustrated in the fourth diagram. Almost all runs have significant outliers of more than 10% of the total execution time compared to the median value of the respective run. However, the distribution of the median values of all runs is very similar, which is remarkable compared to the runtime.

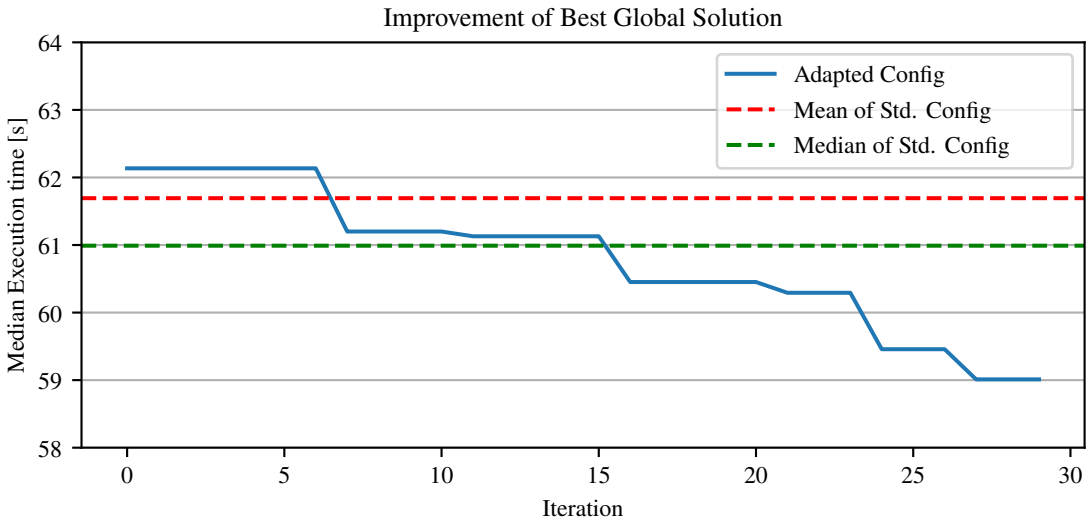

Figure 5.4: Resulting execution times during optimization of binary data processing.

The following conclusion can be drawn from the investigation of the variances of fixed input data and standard Spark configuration:

- Using the execution time of a single execution of a Spark application in a cloud environment with shared and virtualized hardware is not a reliable metric for evaluation. Using several execution times and aggregating their runtime helps to reduce runtime variances; however, only if the runtime differences exceed the threshold of 5%.
- Due to massive outliers that could be observed during some runs using the median value instead of the mean value is the better approach since it's less sensitive toward outliers.
- While the deviations of the median values of the 280 MB and 1.44 GB files are relatively small, the median values of the 90 MB and the 480 MB file differ significantly. This leads to the conclusion that the variance also depends on the runtime and is not static.
- All median values of all investigations lie in a range of variation of +/- 5%. This means that the median value of a run of an improved configuration must fall below this range. Otherwise, whether the execution time results from the improved configuration or external factors leading to a shorter execution time is not distinguishable.

5.3.3 Results of Optimization

The optimization results are illustrated in Figure 5.4. They show a slight but constant improvement over the iterations. The runtime could be lowered from around 62 seconds to 59 seconds, as illustrated in Figure 5.5. This corresponds to an improvement of about 5% compared to the standard Spark configuration. The optimized configuration is shown in Table 5.4. The figure shows the progress of optimization. Whenever the DE algorithm couldn't find an improved configuration, the

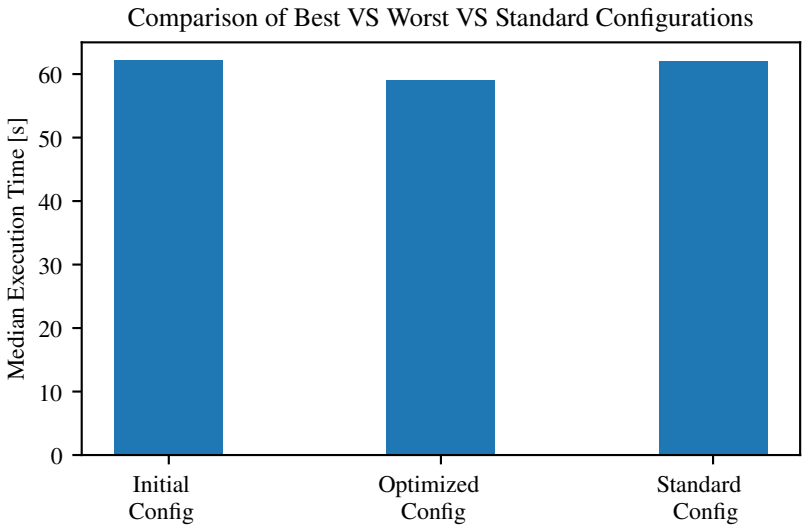

Figure 5.5: Comparison between different sets of configuration and their resulting execution times.

line is steady horizontal, as is the case for the first six iterations. In the following iteration, one population member outperformed the initial best configuration, so the execution time decreased. This phenomenon can also be observed in the following iterations.

5.3.4 Alternative Optimization Goals

Since the absolute execution time is subject to significant variations, different metrics for evaluating a proposed configuration could be considered. These metrics should not depend directly on the absolute execution time but may correlate with it only indirectly. An example of such a metric is the amount of shuffled data in the cluster. Since the data must be transported over the network, this operation requires a lot of overhead and is time-consuming [LTW+15]. This means that any shuffling reduction could also decrease the execution time of a given Spark application. This is examined using the use case of binary data processing.

In Figure 5.6, the variances in shuffling are illustrated that were observed during the conversion of two different file sizes of 90 MB and 280 MB files. The boxplot of the absolute variances of shuffled bytes shows that in both cases, variances are smaller than 1.5 MB. In relation to the shuffled bytes, the relative standard deviation is magnitudes smaller than 1%. It means that the variation of the shuffled bytes is much less than 1% of the mean of shuffled bytes. This makes shuffling very stable and robust as it shows almost no deviation between different executions compared to the execution time. This makes it possible to reduce the number of repetitions of one experiment to only *one*. Again the same parameters are used as presented in Table 5.5. In order to optimize the amount of shuffling the `datagetterfunc()` (Section 4.2) is correspondingly implemented to read out the amount of shuffled bytes as the sum of read and written bytes for all executors.

parameter	dtype	lower_bound	upper_bound	dtype	unit	optimized_val
spark.files.maxPartitionBytes	int	1000000	512000000	int		164669998
spark.reducer.maxSizeInFlight	int	1	500	int	m	346m
spark.shuffle.compress	str_bool	true, false		str_bool		true
spark.shuffle.file.buffer	int	1	500	int	k	65k
spark.network.maxRemoteBlockSizeFetchToMem	int	200	512	int	m	356m
spark.shuffle.spill.compress	str_bool	true, false		str_bool		false
spark.broadcast.blockSize	int	1	500	int	m	234m
spark.broadcast.compress	str_bool	true, false		str_bool		false
spark.broadcast.checksum	str_bool	true, false		str_bool		false
spark.default.parallelism	int	2	16.0	int		6
spark.shuffle.io.maxRetries	int	1	5	int		3
spark.rdd.compress	str_bool	true, false		str_bool		true
spark.serializer	categorical_str	[KryoSerializer, JavaSerializer]		categorical_str		KryoSerializer
spark.io.compression.codec	categorical_str	[snappy, lzf, lz4]		categorical_str		lz4
spark.io.compression.lz4.blockSize	int	2	256	int	k	224k
spark.io.compression.snappy.blockSize	int	2	256	int	k	74k

Table 5.5: Specified parameter space for optimizing the shuffling traffic of binary data processing with the best configuration values found.

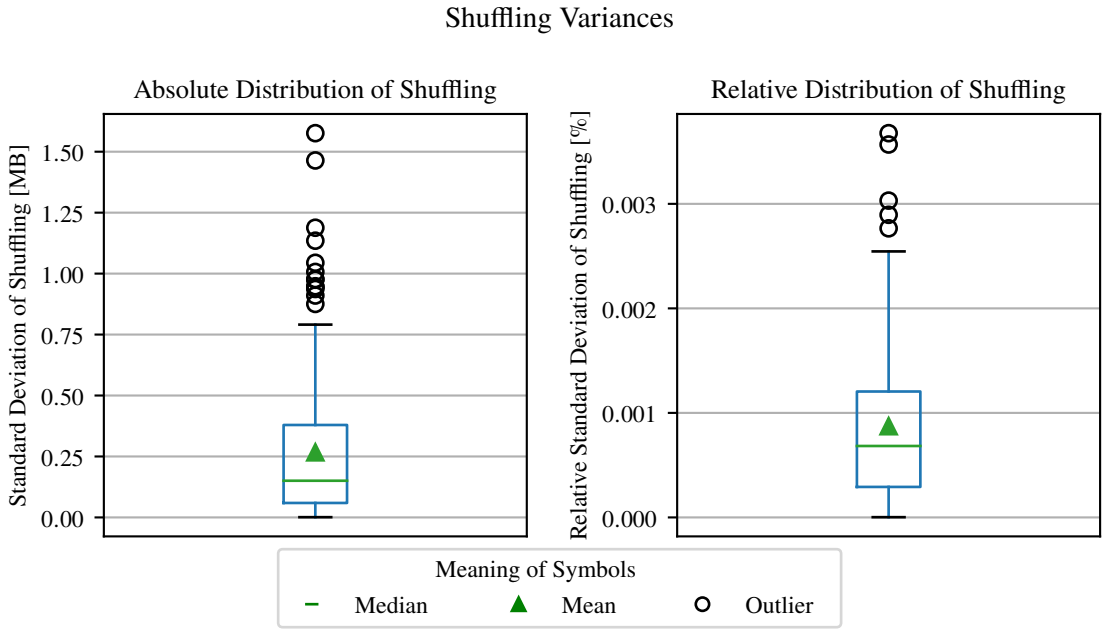

Figure 5.6: Variances of the amount of shuffled bytes using the standard Spark configuration.

The results of optimization of the shuffling traffic of processing a 280 MB file are shown in Figure 5.7. It shows that shuffling could be reduced by roughly 2.5 GB which corresponds to a reduction of roughly 9% compared to the amount of shuffling that the standard Spark configuration causes. The optimized configuration is then tested to evaluate its effect on execution time. The boxplot shown in Figure 5.7 shows the execution time distribution in 10 different runs. The mean and median values lie above those of the ones showing the distribution of execution time shown in the boxplots of Figure 5.3. This indicates that although shuffling could be reduced, it negatively impacts the execution time.

5.4 Optimization of Spark SQL

The second use case where the proposed optimization framework was applied onto falls belongs to Spark SQL. Processing relational data with Spark SQL is a standard feature and covered by the high-level Spark API.

In this exemplary use case, a query on a relational table consisting of 180 GB of data is executed. Several filter, join, and aggregation operations are performed within the query.

Due to the knowledge gains of the first experiment (Section 5.3) the used configuration parameters are slightly changed, and some parameters were set to a fixed value to avoid slow configurations. The configuration parameters used for this experiment are shown in Table 5.8.

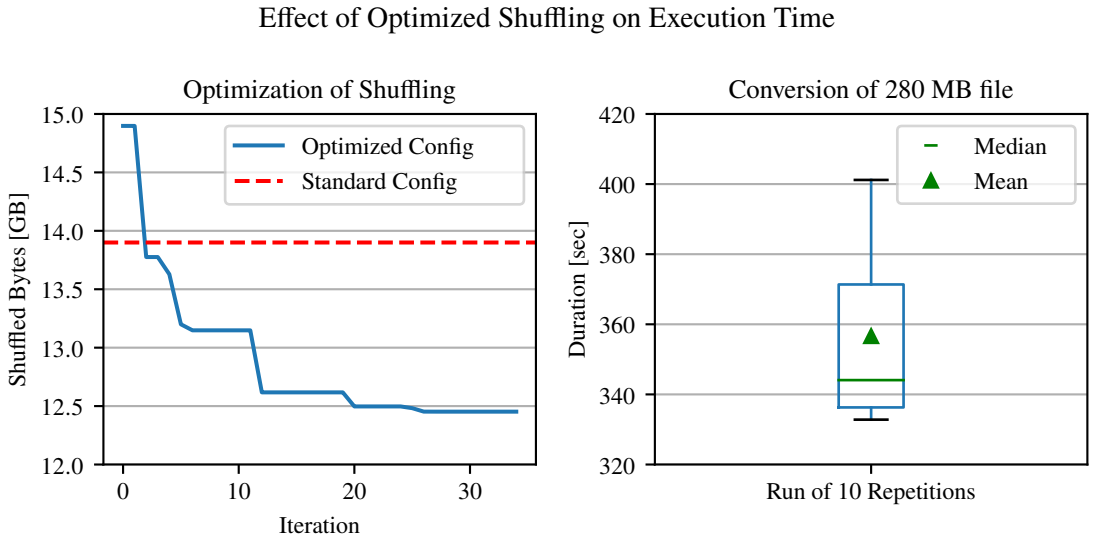

Figure 5.7: Optimization of shuffling and achieved effect on the execution time.

Name of VM Type	No. of Instances	No. of Cores	Memory Size [GB]	Purpose
StandardF4	1	4	4	Spark Driver
Standard_DS3_v2	4	8	8	Spark Workers

Table 5.6: Overview over the used VM types for the SQL optimization.

5.4.1 Design of Experiments

The chosen VM types are shown in Table 5.6. The number of workers has been chosen to be four as a trade-off between performance and costs. The same applies to the type of worker VM, which is also a trade-off between performance and costs. In contrast to the setup in Section 5.3, a greater importance has been given to the amount of memory as the data input sizes are much larger. Therefore, a VM type with a larger memory was chosen.

The configuration of the experiment shows Table 5.7. Due to the higher execution times the number of iterations was reduced.

Hyperparameter	**Value**
Mutation	Randomly drawn from uniform distribution $U(0.5, 1)$ every iteration
Recombination	0.7
Population Size	5
Iterations	15
Number of Repetitions	10

Table 5.7: Overview of the different chosen hyperparameters for optimizing Spark SQL.

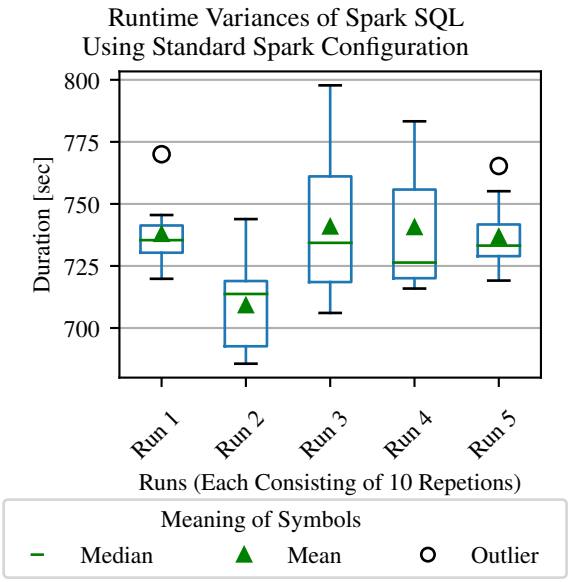

Figure 5.8: Execution time distribution of the Spark-SQL query execution.

Inspired by the work of [XHY22] and the results of the previous use case (Section 5.3), an adapted set of parameters has been selected for the process of optimization and is illustrated in Table 5.8.

5.4.2 Baseline Reference

Several runs using the standard Spark configurations have been performed to evaluate the achieved optimizations. These results then serve to determine a baseline for the comparison. The distributions of the execution times are illustrated in the boxplots of Abbildung 5.8. Each boxplot represents the result of ten executions using the standard Spark configuration and the same input data for the query. The resulting distributions are similar to those in Abbildung 5.3. While most median values are distributed around 730 seconds, the second run has a shorter median execution time. This also means that the runtime variances occur for Spark SQL and must be considered. An optimized configuration must exceed the variance range and be shorter than 690 seconds.

5.4.3 Results of Optimization

In Figure 5.9 the results of applying the framework to the SQL use case are illustrated. The execution times correspond to the global best configuration achieved up to the current iteration. As it can be seen, the framework can optimize the achieved results and find a configuration that results in lower execution times. The initial start time could be improved by around 50 seconds, corresponding to an improvement of about 6% compared to the first iteration. Compared with the results of Figure 5.8, it is obvious that the execution times using the standard Spark configuration

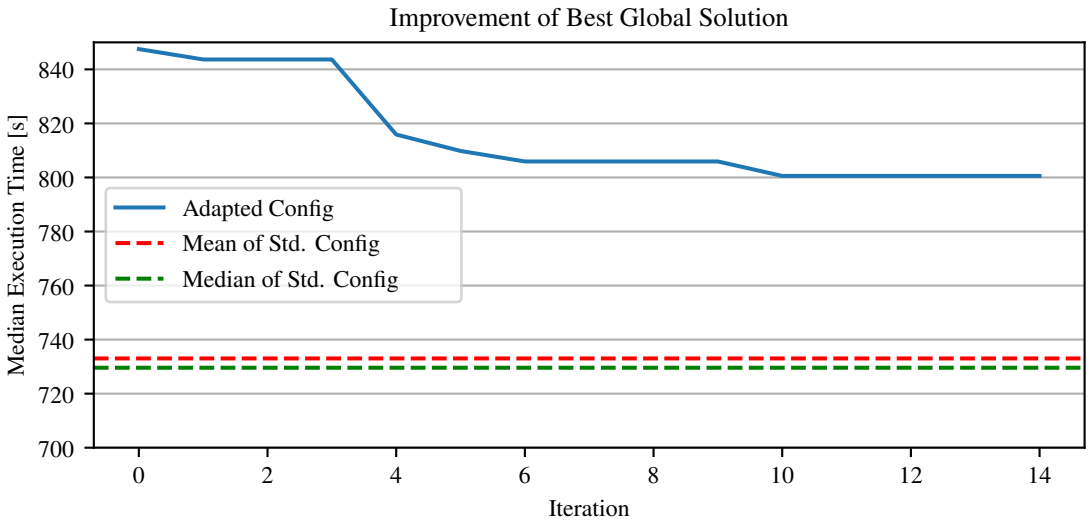

Figure 5.9: Obtained results when the framework is applied to search for optimized Spark SQL configurations.

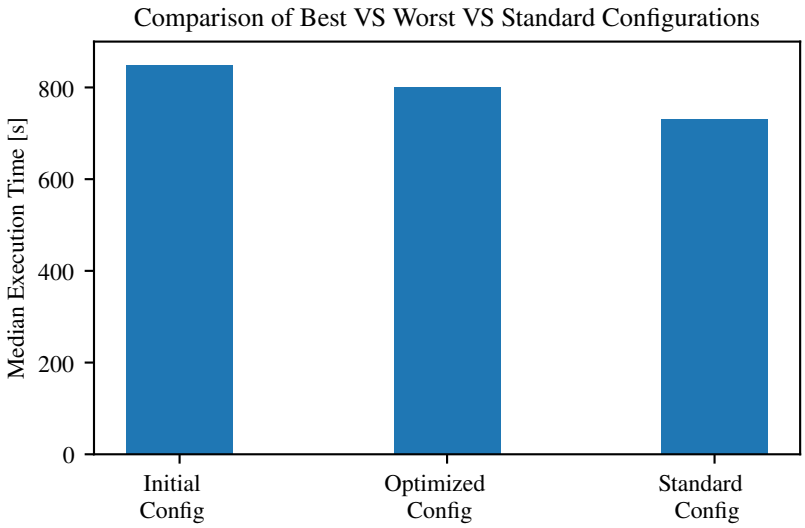

Figure 5.10: Comparison between different sets of Spark SQL configurations and their resulting execution times.

still achieve a significantly shorter execution time, as Figure 5.10 illustrates. This means that no configuration was found that outperformed the standard Spark configuration. Potential reasons for this phenomenon are discussed in Chapter 6.

parameter	lower_bound	upper_bound	dtype	unit	optimized_val
spark.executor.cores	1	4	int		2
spark.executor.memory	1000	7284	int	m	2051m
spark.files.maxPartitionBytes	1000000	512000000	int		207456383
spark.memory.fraction	0.5	0.9	float		0.81
spark.memory.storageFraction	0.5	0.9	float		0.84
spark.reducer.maxSizeInFlight	1	500	int	m	443m
spark.shuffle.file.buffer	1	500	int	k	295k
spark.network.maxRemoteBlockSizeFetchToMem	200	512	int	m	453m
spark.broadcast.blockSize	1	500	int	m	21m
spark.io.compression.lz4.blockSize	2	256	int	k	32k
spark.sql.shuffle.partitions	16	64	int		53
spark.sql.adaptive.autoBroadcastJoinThreshold	0	100000000	int		24191943
spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold	0	100000000	int		69439385

Table 5.8: Specified parameter space for optimizing Spark SQL execution time, showing the best parameter values.

6 Discussion

This chapter discusses the achieved results. The discussion includes the framework, results, and observations from the performed experiments.

6.1 Discussion of the Case Study

The results of the performed experiments bring some vital information that was unexpected. One of the main challenges discovered during this work was inconsistency in execution times. Executing the same Spark application several times with fixed input data and fixed configuration can still lead to various runtime variances. A potential explanation for this observation is the virtualized Spark cluster, consisting of virtualized machines in a heterogeneous cloud environment. This effect may be explained by external effects like high usage of the physical machines which host the virtualized Spark nodes. Another reason for the observed variance may be increased local network use or high load on the file storage where input data is read from and results are stored. Since that information is not available to the standard applicant of Databricks, determining the exact reason is not possible.

Repeating the execution of a chosen Spark configuration reduces the impact of the high runtime variances but does not solve them completely.

The case studies in Chapter 5 have shown that even when using several runs, the resulting mean, as well as the median value, can differ significantly. This makes it especially difficult to determine which execution time should be used for the evaluation. In certain circumstances, a potentially optimized configuration may seem way better performing by randomly choosing a run of standard configurations with a set of long runtimes. When evaluated in detail, this effect becomes then apparent. During the process of optimization, however, detecting this effect is challenging. One potential solution to this problem would be to increase the number of repetitions further. This approach leads to higher execution costs, especially for longer-running applications, and makes the approach problematic. Determining the number of sufficient repetitions may also be challenging. This effort may only be feasible for repetitive, long-running tasks where also minor improvements can have a financial impact. However, for long-running Spark applications, searching for an optimized configuration with numerous repetitions of each configuration can quickly become infeasible.

For the first use case of binary data processing, an improvement of roughly 5% could be achieved. However, no significant improvement in execution time could be achieved for the second use case that outperformed the provided standard Spark configuration. Nevertheless, considering that the

optimized configurations are optimized for the specific use case while the standard configuration is generally applicable and still offers good performance, the change of values in the standard configuration should be well overthought.

Another realization of the experiments is that the configuration specified by the Databricks platform was already suitable for both exemplary use cases. Only minor improvements could be achieved in the case of binary data processing. The second use case didn't bring improvements at all.

Changing the configuration only makes sense for cases where the Spark cluster fails due to the standard configuration not providing enough resources for the cluster. The second kind of application where Spark configuration optimization may be considered is long-running or repetitive Spark applications.

6.1.1 Databricks Runtime Environment

The reason why many other publications achieve much better optimization results than those of this work could lie in the runtime environment. As already illustrated in Chapter 5, high runtime variances are observable and have a strong impact. Due to the use of VMs in the cloud environment and the provided cloud storage, the provided hardware resources are shared with other cloud customers and cannot be controlled. Therefore, most other works use an experimental setup where the provided resources are exclusively dedicated to the Spark cluster [CYWL21; PPCB20; XHY22]. This eliminates the need for many repetitions and makes the results much more reliable. Unfortunately, the only other work [CYWL21] that observes runtime variances no longer mentions them for discussing the achieved results. However, since many Spark applicants use such a runtime environment and not an exclusive server infrastructure, the area of optimization in those runtime variances is still of interest despite the significant challenges it brings. Therefore, future works should focus on building more time robust metrics and methods to transform or normalize the achieved results to reduce the time variance or eliminate it.

Another explanation for the lower optimization results lies in the Databricks settings themselves. Contrary to the standard Spark configuration, the Databricks platform knows which hardware type for the Spark cluster was selected. It can hence adapt some parameters to the VM type contrary to the standard Spark configuration, like the memory provided to the executors. The standard configuration specified by Databricks assigns all available cores to an executor and all the available memory compared to only 1 GB per executor in the Spark default configuration.

This may explain why other works achieved such high-performance improvements in contrast to this work.

6.2 Observed Correlations

During experiments, several investigations and analyses have been performed to evaluate the effect and the correlation of some parameters on the optimized target.

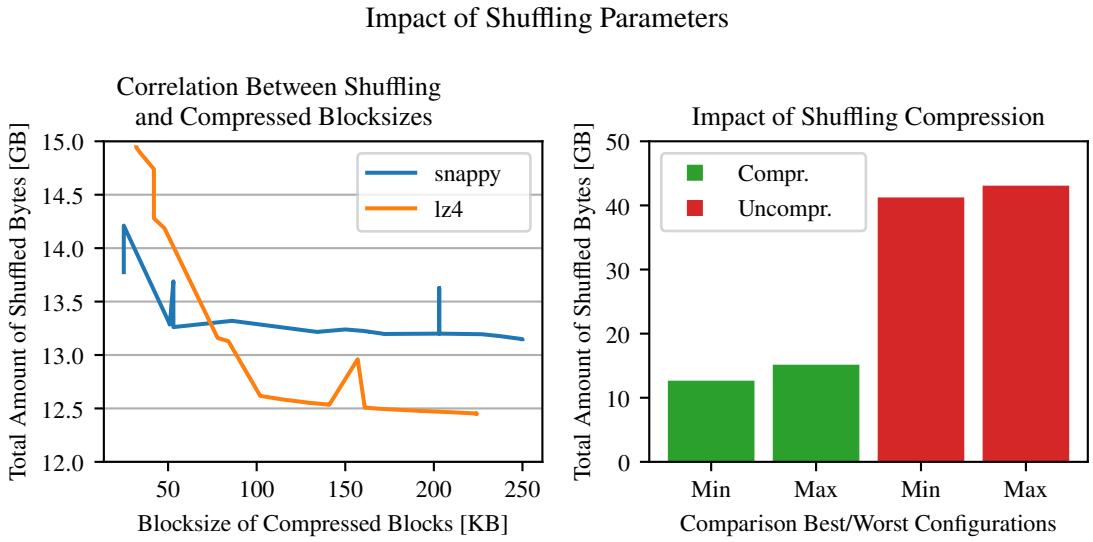

Figure 6.1: Impact of shuffling block size and shuffling compression.

In the case of optimizing the shuffled amount of bytes within the Spark cluster, some observations could be made. The most significant impact on the number of shuffled bytes is, unsurprisingly, the option of whether to compress the shuffling or not. Apart from this evident parameter influence, two other influential parameters were identified, namely `spark.io.compression.codec` and `spark.io.compression.lz4.blockSize`. The former selects the compression codec used to compress the shuffling, and the latter determines the block size where the compressed bytes are stored. The experiments revealed that the `lz4` compression codec produces the lowest amount of shuffled bytes, i.e., it is highly efficient. The `lz4` codec is already the selected standard compression algorithm.

While these findings are expected, an unexpected result was the negative correlation between the block size and the shuffled amount of bytes (Figure 6.1). For larger block sizes, the amount of shuffled bytes decreases. A potential explanation for this effect could be that more compressed data can be placed into the block for larger block sizes. However, the experiments have shown that execution time couldn't be reduced despite the reduced shuffling traffic. This indicates that a too large block size may be expensive to create or to send via the network.

6.3 Evaluating the Optimization Framework

In this section, the framework itself is discussed. This includes the ability of future framework extensions and a discussion of the empirical approach.

6.3.1 Extensibility

One of the core design principles of the optimization framework was to focus on extensibility and the ability to adapt to future requirements and application scenarios. Due to this reason, the framework is realized by depending on interfaces and abstract classes (Figure 4.1). Hence the main components of the framework can be adapted in the future. In the future, new search algorithms could be examined easily without significant changes to the framework's design.

A new search algorithm only needs to implement the corresponding interface, and the other components remain unchanged. Such new search algorithms could be *Particle Swarm Optimization* [KE95] or *Bayesian Optimization* [Moc89] or other search algorithms that are promising in the area of Spark configuration optimization.

If the intended initial runtime environment needs to be changed, the adaptions to the framework are also minimal. In this case, only two components need to be adapted. To those components belong the one that implements the *ExperimentRunnerFactory* interface and the one that inherits from the abstract *AbstractExperimentRunner* class. This ensures that the necessary application logic is implemented. The new and adapted components can implement the necessary steps to execute the proposed configurations. Such new runtime environments could be a "bare-metal" cluster or an oracle-based approach to predict the objective function value. It is thus avoided to perform costly experiments. The approach is described in detail Section 7.2.

The component that is responsible to evaluate the collected statistics must implement the *StatisticsEvaluatorInterface*. This allows for new and complex evaluation approaches and new metrics to be used while the other components remain unchanged.

The use of interfaces and abstract classes makes the framework generally adaptable and extendable. This makes it possible to use the framework theoretically for any optimization tasks by adapting the relevant components.

6.3.2 Discussion of the Empirical Approach

The most important advantage of the empirical approach is that it uses real Spark applications. This means that no information is lost due to any kind of modeling and abstraction of the real Spark applications. However, this approach also has disadvantages.

Since the actual Spark application must be executed, long-running Spark applications will cause the same costs as if they run in production. This means that the empirical approach will cause high optimization effort and cost if many different configurations should be examined. This problem increases in the case of an environment with high runtime variances. Here, many repetitions must be executed to find the true execution time statistically.

In the case of ad-hoc execution of Spark applications, the optimization may be unfeasible. This is due to the nature of the empirical black box approach. The search and examination of the different configurations may be much longer than the time of executing the application itself. However, since

it's an ad-hoc application, it won't be executed again, meaning that for any new ad-hoc application, this process has to be repeated. The optimization process causes more effort compared to the single execution of the application itself.

Even if an improved configuration is found, the question arises: What happens if the size of input data changes? This could result in executing the optimization process again, causing a high overhead.

Future works should address the high effort and costs of the optimization. Some proposals are made in Section 7.2.

7 Conclusion

This chapter briefly sums up the results of this work. Finally, an outlook for future work in optimizing Spark configurations concludes this chapter.

7.1 Summary of Results

In this work, a framework for optimizing Spark configurations has been presented. Its practicability could be demonstrated in a case study. Applicants of the framework can conveniently specify the optimization parameters of interest. Apart from specifying some additional hyperparameters, this is enough to start the optimization process. The flexible implementation of the framework allows for quickly setting up the framework for different Spark applications. Due to the framework's extensibility, new search and optimization algorithms can be used while the effort of adaption is minimal. The investigations and experiments performed in this work have shown the challenges of optimizing Spark configurations in a heterogeneous cloud environment. Cloud environments offer typically virtualized, shared hardware resulting in a varying performance of the VMs. Additionally to the actual load on the physical hardware, other factors like the load on the network or the load on the storage can lead to significant and unforeseen changes in the runtime characteristics of Spark applications with unchanged settings. This makes the optimization process of the execution time difficult since the difference between a real improvement and a positive external influence in the runtime characteristics is hard to detect.

Different approaches have been presented to mitigate those effects, like using statistics to reduce variance or using alternative optimization goals that correlate only indirectly with the execution time. While they solve some of the issues, they cannot fully mitigate them.

7.2 Outlook and Future Work

Some primary challenges have been identified that require further investigation and work. One of those challenges is the significant effort of a purely empirical approach to optimize the Spark configurations. This can lead to long execution times in the case of complex applications. The effect is even worsened due to the heterogeneous cloud environments. Several repetitions must be executed per configuration to get a more reliable execution time. Since typically, a cloud provider bills its customer based on the time they used the rented resources, repetitive and long-running applications cause especially high costs.

One of the approaches to avoiding the need to execute the Spark application is to use ML based models that can predict the execution time for a given Spark application or given execution characteristics. A substantially large set of training data is needed to train the models. In this case, an efficient empirical approach like the one proposed in this work can be used to generate this data. Since the search algorithm of the proposed framework tries to search for improved configurations, this can teach the ML models the correlation between parameters and how to improve them.

A future extension of the framework could be to add an offline phase in which a model is trained by the generated data and can then be used as an oracle to predict the execution time of the Spark application. This can reduce the required number of actual executions of the Spark cluster. In addition, the process of querying the oracle is magnitudes faster compared with the execution of real Spark applications, and thus more configuration candidates can be evaluated.

An essential aspect of such models would be taking input data sizes of Spark applications into account and training on generic features to avoid tight coupling to a particular application.

The costs of collecting the training data could be significantly reduced in environments where lots of data is processed continuously. Instead of executing dedicated and expensive experiments, the execution statistics and information could be collected from the Spark clusters used in production. A central data store could serve as storage for the relevant statistics. These statistics can be used to train the ML model. This approach could drastically reduce the cost of training because the number of experiments could be reduced if not omitted entirely.

The third area of future work is the variance in runtimes. This work has shown that optimizing Spark configurations of applications used in practice in virtualized cloud runtime environments is challenging. Therefore, time robust metrics should be designed to reduce the impact of runtime variances. This would not only help to guide an optimizer towards a better configuration but also reduce the number of repetitions needed for the statistical aggregation. The advantage would be a reduction in optimization cost or allowing to try out more configuration candidates at the same cost.

Bibliography

[AAh19] Z. A. Al-Sai, R. Abdullah, M. heikal husin. "Big Data Impacts and Challenges: A Review". In: *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT). Amman, Jordan: IEEE, Apr. 2019, pp. 150–155. ISBN: 978-1-5386-7942-5. DOI: 10.1109/JEEIT.2019.8717484. URL: https://ieeexplore.ieee.org/document/8717484/ (visited on 08/12/2022) (cit. on p. 15).

[Agr22] Agrawroh. *Databricks*. In: *Wikipedia*. Sept. 19, 2022. URL: https://en.wikipedia.org/w/index.php?title=Databricks&oldid=1111229831 (visited on 09/30/2022) (cit. on p. 54).

[ALA+19] C. Alexopoulos, Z. Lachana, A. Androutsopoulou, V. Diamantopoulou, Y. Charalabidis, M. A. Loutsaris. "How Machine Learning Is Changing E-Government". In: *Proceedings of the 12th International Conference on Theory and Practice of Electronic Governance*. ICEGOV2019: 12th International Conference on Theory and Practice of Electronic Governance. Melbourne VIC Australia: ACM, Apr. 3, 2019, pp. 354–363. ISBN: 978-1-4503-6644-1. DOI: 10.1145/3326365.3326412. URL: https://dl.acm.org/doi/10.1145/3326365.3326412 (visited on 09/27/2022) (cit. on p. 15).

[Bas22] D. Bassermann. *Home*. Jan. 1, 2022. URL: https://www.asam.net/ (visited on 09/15/2022) (cit. on p. 28).

[BME19] L. Bernardi, T. Mavridis, P. Estevez. "150 Successful Machine Learning Models: 6 Lessons Learned at Booking.Com". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19: The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Anchorage AK USA: ACM, July 25, 2019, pp. 1743–1751. ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330744. URL: https://dl.acm.org/doi/10.1145/3292500.3330744 (visited on 09/27/2022) (cit. on p. 15).

[Bor22] D. Borthakur. *HDFS Architecture Guide*. HDFS Architecture Guide. May 18, 2022. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (visited on 10/01/2022) (cit. on p. 23).

[BYL+18] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, S. Feng. "Configuring In-Memory Cluster Computing Using Random Forest". In: *Future Generation Computer Systems* 79 (Feb. 2018), pp. 1–15. ISSN: 0167739X. DOI: 10.1016/j.future.2017.08.011. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X16305167 (visited on 09/02/2022) (cit. on p. 16).

[Cor22] M. Corporation. *Microsoft Azure*. In: *Wikipedia*. Sept. 4, 2022. URL: https://de.wikipedia.org/w/index.php?title=Microsoft_Azure&oldid=225900951 (visited on 09/30/2022) (cit. on p. 54).

[CSG+18] Z. Chao, S. Shi, H. Gao, J. Luo, H. Wang. "A Gray-Box Performance Model for Apache Spark". In: *Future Generation Computer Systems* 89 (Dec. 2018), pp. 58–67. ISSN: 0167739X. DOI: 10.1016/j.future.2018.06.032. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X17323233 (visited on 04/15/2022) (cit. on pp. 16, 30).

[CWL+20] Y. Cheng, C. Wu, Y. Liu, R. Ren, H. Xu, B. Yang, Z. Qi. "OPS: Optimized Shuffle Management System for Apache Spark". In: *49th International Conference on Parallel Processing - ICPP*. ICPP '20: 49th International Conference on Parallel Processing. Edmonton AB Canada: ACM, Aug. 17, 2020, pp. 1–11. ISBN: 978-1-4503-8816-0. DOI: 10.1145/3404397.3404430. URL: https://dl.acm.org/doi/10.1145/3404397.3404430 (visited on 04/23/2022) (cit. on p. 22).

[CYW21] G. Cheng, S. Ying, B. Wang. "Tuning Configuration of Apache Spark on Public Clouds by Combining Multi-Objective Optimization and Performance Prediction Model". In: *Journal of Systems and Software* 180 (Oct. 2021), p. 111028. ISSN: 01641212. DOI: 10.1016/j.jss.2021.111028. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121221001254 (visited on 04/16/2022) (cit. on pp. 16, 30, 35).

[CYWL21] G. Cheng, S. Ying, B. Wang, Y. Li. "Efficient Performance Prediction for Apache Spark". In: *Journal of Parallel and Distributed Computing* 149 (Mar. 2021), pp. 40–51. ISSN: 07437315. DOI: 10.1016/j.jpdc.2020.10.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731520303993 (visited on 04/15/2022) (cit. on pp. 16, 23, 29, 78).

[CZ18] B. Chambers, M. Zaharia. *Spark: The Definitive Guide: Big Data Processing Made Simple*. First edition. Sebastapol, CA: O'Reilly Media, 2018. 576 pp. ISBN: 978-1-4919-1221-8 (cit. on p. 19).

[Dat22a] Databricks. *Data Lakehouse Architecture and AI Company*. Databricks. Jan. 1, 2022. URL: https://www.databricks.com/ (visited on 09/09/2022) (cit. on p. 24).

[Dat22b] Databricks. *REST API (Latest) | Databricks on AWS*. Jan. 1, 2022. URL: https://docs.databricks.com/dev-tools/api/latest/index.html (visited on 09/20/2022) (cit. on pp. 24, 38, 53).

[FKE+15] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf (cit. on p. 15).

[Fou18] A. S. Foundation. *Apache Spark™ - Unified Engine for Large-Scale Data Analytics*. Jan. 1, 2018. URL: https://spark.apache.org/ (visited on 09/09/2022) (cit. on p. 19).

[Fou22a] A. S. Foundation. *Apache Cassandra | Apache Cassandra Documentation*. Jan. 1, 2022. URL: https://cassandra.apache.org/_/index.html (visited on 09/09/2022) (cit. on p. 23).

[Fou22b] A. S. Foundation. *Apache Spark*. In: *Wikipedia*. Mar. 18, 2022. URL: https://de.wikipedia.org/w/index.php?title=Apache_Spark&oldid=221272237 (visited on 09/30/2022) (cit. on p. 54).

[Fou22c] A. S. Foundation. *Cluster Mode Overview - Spark 3.3.0 Documentation*. Jan. 1, 2022. URL: https://spark.apache.org/docs/latest/cluster-overview.html (visited on 09/09/2022) (cit. on pp. 19, 20, 23).

[Fou22d] A. S. Foundation. *Configuration - Spark 3.3.0 Documentation*. Jan. 1, 2022. URL: https://spark.apache.org/docs/latest/configuration.html (visited on 09/09/2022) (cit. on pp. 23, 35).

[Fou22e] A. S. Foundation. *Tuning - Spark 3.3.0 Documentation*. Jan. 1, 2022. URL: https://spark.apache.org/docs/latest/tuning.html (visited on 09/09/2022) (cit. on p. 23).

[GT18] A. Gounaris, J. Torres. "A Methodology for Spark Parameter Tuning". In: *Big Data Research* 11 (Mar. 2018), pp. 22–32. ISSN: 22145796. DOI: 10.1016/j.bdr.2017.05.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S2214579617300114 (visited on 09/02/2022) (cit. on p. 16).

[HCL21] H. Herodotou, Y. Chen, J. Lu. "A Survey on Automatic Parameter Tuning for Big Data Processing Systems". In: *ACM Computing Surveys* 53.2 (Mar. 31, 2021), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3381027. URL: https://dl.acm.org/doi/10.1145/3381027 (visited on 05/08/2022) (cit. on pp. 15, 16, 31).

[HHL18] X. Hua, M. C. Huang, P. Liu. "Hadoop Configuration Tuning With Ensemble Modeling and Metaheuristic Optimization". In: *IEEE Access* 6 (2018), pp. 44161–44174. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2857852. URL: https://ieeexplore.ieee.org/document/8416665/ (visited on 09/29/2022) (cit. on p. 33).

[HXLL17] S. Huang, J. Xu, R. Liu, H. Liao. "A Novel Compression Algorithm Decision Method for Spark Shuffle Process". In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017 IEEE International Conference on Big Data (Big Data). Dec. 2017, pp. 2931–2940. DOI: 10.1109/BigData.2017.8258262 (cit. on p. 22).

[Ili20] R. Ilijason. *Beginning Apache Spark Using Azure Databricks: Unleashing Large Cluster Analytics in the Cloud*. Berkeley, CA: Apress, 2020. ISBN: 978-1-4842-5780-7 978-1-4842-5781-4. DOI: 10.1007/978-1-4842-5781-4. URL: http://link.springer.com/10.1007/978-1-4842-5781-4 (visited on 04/30/2022) (cit. on p. 24).

[JM15] M. I. Jordan, T. M. Mitchell. "Machine Learning: Trends, Perspectives, and Prospects". In: *Science* 349.6245 (July 17, 2015), pp. 255–260. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aaa8415. URL: https://www.science.org/doi/10.1126/science.aaa8415 (visited on 09/27/2022) (cit. on p. 15).

[Jup22] Jupyter. *Project Jupyter*. jupyter. Jan. 1, 2022. URL: https://jupyter.org (visited on 10/01/2022) (cit. on p. 24).

[KE95] J. Kennedy, R. Eberhart. "Particle Swarm Optimization". In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Proceedings of ICNN'95 - International Conference on Neural Networks. Vol. 4. Nov. 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968 (cit. on p. 80).

[KKL10] A. E. Khandani, A. J. Kim, A. W. Lo. "Consumer Credit-Risk Models via Machine-Learning Algorithms". In: *Journal of Banking & Finance* 34.11 (Nov. 2010), pp. 2767–2787. ISSN: 03784266. DOI: 10.1016/j.jbankfin.2010.06.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0378426610002372 (visited on 09/27/2022) (cit. on p. 15).

[LTW+15] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura. "SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. CF'15: Computing Frontiers Conference. Ischia Italy: ACM, May 6, 2015, pp. 1–8. ISBN: 978-1-4503-3358-0. DOI: 10.1145/2742854.2747283. URL: https://dl.acm.org/doi/10.1145/2742854.2747283 (visited on 04/17/2022) (cit. on pp. 22, 31, 68).

[mag22] m. magicheron. *AzureDataLakeGen2-SDK*. Feb. 14, 2022. URL: https://github.com/magicheron/AzureDataLakeGen2-SDK (visited on 09/30/2022) (cit. on p. 54).

[Mic22] Microsoft. *Virtuelle Linux-Computer – Preise | Microsoft Azure*. Jan. 1, 2022. URL: https://azure.microsoft.com/de-de/pricing/details/virtual-machines/linux/ (visited on 09/14/2022) (cit. on p. 63).

[MOC18] A. Mosavi, P. Ozturk, K.-w. Chau. "Flood Prediction Using Machine Learning Models: Literature Review". In: *Water* 10.11 (Oct. 27, 2018), p. 1536. ISSN: 2073-4441. DOI: 10.3390/w10111536. URL: https://www.mdpi.com/2073-4441/10/11/1536 (visited on 09/27/2022) (cit. on p. 15).

[Moc89] J. Mockus. *Bayesian Approach to Global Optimization: Theory and Applications*. Red. by M. Hazewinkel. Vol. 37. Mathematics and Its Applications. Dordrecht: Springer Netherlands, 1989. ISBN: 978-94-010-6898-7 978-94-009-0909-0. DOI: 10.1007/978-94-009-0909-0. URL: http://link.springer.com/10.1007/978-94-009-0909-0 (visited on 09/06/2022) (cit. on p. 80).

[Mon22] MongoDB. *MongoDB: The Developer Data Platform*. MongoDB. Jan. 1, 2022. URL: https://www.mongodb.com (visited on 09/09/2022) (cit. on p. 23).

[NMXS21] D. Nikitopoulou, D. Masouros, S. Xydis, D. Soudris. "Performance Analysis and Auto-tuning for SPARK in-Memory Analytics". In: *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2021 Design, Automation Test in Europe Conference Exhibition (DATE). Feb. 2021, pp. 76–81. DOI: 10.23919/DATE51398.2021.9474122 (cit. on pp. 16, 29).

[PGT17] P. Petridis, A. Gounaris, J. Torres. "Spark Parameter Tuning via Trial-and-Error". In: *Advances in Big Data*. Ed. by P. Angelov, Y. Manolopoulos, L. Iliadis, A. Roy, M. Vellasco. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2017, pp. 226–237. ISBN: 978-3-319-47898-2. DOI: 10.1007/978-3-319-47898-2_24 (cit. on pp. 16, 30).

[PPCB20] D. B. Prats, F. A. Portella, C. H. A. Costa, J. L. Berral. "You Only Run Once: Spark Auto-Tuning From a Single Run". In: *IEEE Transactions on Network and Service Management* 17.4 (Dec. 2020), pp. 2039–2051. ISSN: 1932-4537, 2373-7379. DOI: 10.1109/TNSM.2020.3034824. URL: https://ieeexplore.ieee.org/document/9244226/ (visited on 04/14/2022) (cit. on pp. 28, 78).

[PRGK14] I. Polato, R. Ré, A. Goldman, F. Kon. "A Comprehensive View of Hadoop Research—A Systematic Literature Review". In: *Journal of Network and Computer Applications* 46 (Nov. 2014), pp. 1–25. ISSN: 10848045. DOI: 10.1016/j.jnca.2014.07.022. URL: https://linkinghub.elsevier.com/retrieve/pii/S1084804514001635 (visited on 08/12/2022) (cit. on p. 15).

[Ros60] H. H. Rosenbrock. "An Automatic Method for Finding the Greatest or Least Value of a Function". In: *The Computer Journal* 3.3 (Mar. 1, 1960), pp. 175–184. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/3.3.175. URL: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/3.3.175 (visited on 08/29/2022) (cit. on p. 57).

[SBB20] F. Schiavio, D. Bonetta, W. Binder. "Dynamic Speculative Optimizations for SQL Compilation in Apache Spark". In: *Proceedings of the VLDB Endowment* 13.5 (Jan. 2020), pp. 754–767. ISSN: 2150-8097. DOI: 10.14778/3377369.3377382. URL: https://dl.acm.org/doi/10.14778/3377369.3377382 (visited on 09/02/2022) (cit. on pp. 15, 19).

[SDC+16] S. Salloum, R. Dautov, X. Chen, P. X. Peng, J. Z. Huang. "Big Data Analytics on Apache Spark". In: *International Journal of Data Science and Analytics* 1.3 (Nov. 1, 2016), pp. 145–164. ISSN: 2364-4168. DOI: 10.1007/s41060-016-0027-9. URL: https://doi.org/10.1007/s41060-016-0027-9 (visited on 04/16/2022) (cit. on pp. 15, 19–21).

[SP97] R. Storn, K. Price. "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces". In: *Journal of Global Optimization* 11.4 (Dec. 1, 1997), pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328. URL: https://doi.org/10.1023/A:1008202821328 (cit. on pp. 26, 27, 35, 43).

[VGO+20] P. Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17.3 (Mar. 2, 2020), pp. 261–272. ISSN: 1548-7091, 1548-7105. DOI: 10.1038/s41592-019-0686-2. URL: http://www.nature.com/articles/s41592-019-0686-2 (visited on 09/12/2022) (cit. on pp. 43, 47).

[Wey21] D. Weyns. *An Introduction to Self-Adaptive Systems a Contemporary Software Engineering Perspective*. 2021. ISBN: 978-1-119-57493-4 978-1-119-57491-0. URL: https://doi.org/10.1002/9781119574910 (visited on 08/04/2022) (cit. on pp. 24–26).

[XHY22] J. Xin, K. Hwang, Z. Yu. “LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD/PODS ’22: International Conference on Management of Data. Philadelphia PA USA: ACM, June 10, 2022, pp. 674–684. ISBN: 978-1-4503-9249-5. DOI: 10.1145/3514221.3526157. URL: https://dl.acm.org/doi/10.1145/3514221.3526157 (visited on 08/13/2022) (cit. on pp. 16, 31, 72, 78).

[XX14] Xue-Wen Chen, Xiaotong Lin. “Big Data Deep Learning: Challenges and Perspectives”. In: *IEEE Access* 2 (2014), pp. 514–525. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2325029. URL: http://ieeexplore.ieee.org/document/6817512/ (visited on 09/27/2022) (cit. on p. 15).

[ZCD+12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: (Apr. 25, 2012), p. 14 (cit. on pp. 15, 20).

[ZCF+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. “Spark: Cluster Computing with Working Sets”. In: (June 22, 2010), p. 7 (cit. on p. 20).

[ZCS+18] H. Zhang, B. Cho, E. Seyfe, A. Ching, M. J. Freedman. “Riffle: Optimized Shuffle Service for Large-Scale Data Analytics”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18: Thirteenth EuroSys Conference 2018. Porto Portugal: ACM, Apr. 23, 2018, pp. 1–15. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190534. URL: https://dl.acm.org/doi/10.1145/3190508.3190534 (visited on 04/23/2022) (cit. on pp. 15, 22).

[ZLW18] H. Zhang, Z. Liu, L. Wang. “Tuning Performance of Spark Programs”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018 IEEE International Conference on Cloud Engineering (IC2E). Orlando, FL: IEEE, Apr. 2018, pp. 282–285. ISBN: 978-1-5386-5008-0. DOI: 10.1109/IC2E.2018.00057. URL: https://ieeexplore.ieee.org/document/8360342/ (visited on 04/19/2022) (cit. on p. 21).

[ZSK15] J. Zakir, T. Seymour, B. Kristi. “BIG DATA ANALYTICS”. In: *Issues In Information Systems* (2015). ISSN: 15297314. DOI: 10.48009/2_iis_2015_81-90. URL: https://iacis.org/iis/2015/2_iis_2015_81-90.pdf (visited on 09/01/2022) (cit. on p. 15).

A Appendix

A.1 Domain and Parameter Specification Structure

The following figures show the expected format of the domain and parameter specifications. They show required and optional fields.

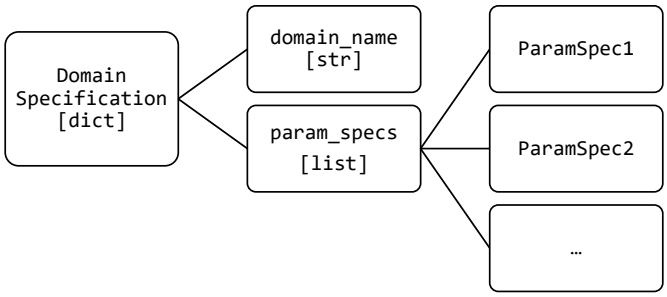

Figure A.1: Expected key-value structure of a specified domain.

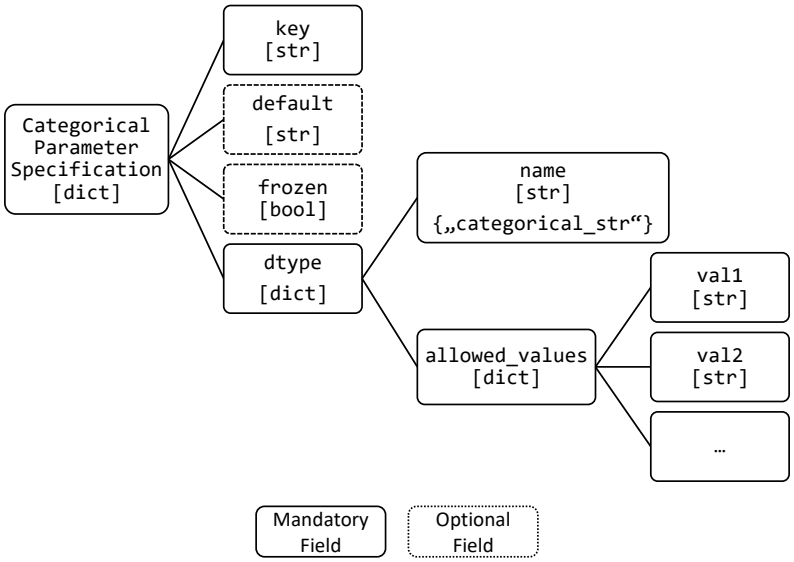

Figure A.2: Expected key-value structure of a categorical parameter specification.

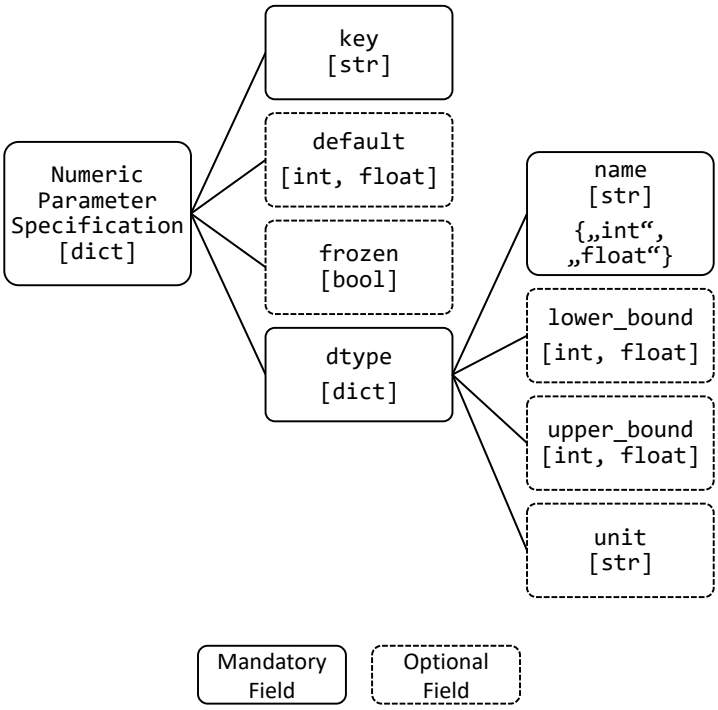

Figure A.3: Expected key-value structure of a numerical parameter specification.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature