



Faculty of Science and Technology

MASTER THESIS

Study program/Specialization: Spring semester 2023

M.Sc. Data Science

Open

Authors: Håvard Moe Jacobsen, Ola André Flotve

Faculty supervisor: Rui Paulo Maximo Pereira Mateus Esteves

Master Thesis Title:

Spark Optimization: A Column Recommendation System for
Data Partitioning and Z-Ordering on ETL Platforms

Credits (ECTS): 30

Keywords:

Spark Optimization, Partitioning,
Z-Ordering, Spark Event Logs,
Column Recommendation System,
ETL Platform

Pages: 75

Stavanger 15. June 2023

"If we knew what it was we were doing, it would not be called research, would it?"

— Albert Einstein

Abstract

In this thesis, we present a solution for the challenge of optimizing the retrieval of data in Spark. Our column recommendation system is based on Spark's event logs and finds influential columns for Z-ordering and partitioning. The column recommendation system consists of four methods, each looking for different query patterns and query characteristics. From the recommendation system experiment, we managed to improve the run time by 17% compared to the baseline. This improvement demonstrates our column recommendation system's potential for optimizing data retrieval in Spark. Our system was developed on an ETL platform and is a flexible solution for ETL platforms utilizing Spark.

Acknowledgements

We want to thank Bouvet and Equinor for providing us with the opportunity to write this thesis. We would also like to thank our supervisor: Rui Paulo Maximo Pereira Mateus Esteves, for his expertise and support throughout this research. His mentorship and concrete feedback were important when shaping the direction and quality of the thesis.

Contents

Abstract	v
Acknowledgments	vii
Contents	ix
Sammendrag	xi
1 Introduction	1
1.1 Motivation	1
1.2 Problem Definition	1
1.3 Use Case	2
1.4 Research Questions	2
1.5 Challenges	2
1.6 Outline	4
2 Background	5
2.1 ETL	5
2.2 MapReduce	6
2.3 Apache Spark	7
2.3.1 Architecture	7
2.3.2 APIs in Spark	9
2.3.3 Physical Plan	13
2.3.4 The Catalyst Optimizer	14
2.3.5 Spark UI	17
2.3.6 Spark Event Logs	18
2.4 Apache Parquet	20
2.5 Azure Data Lake	21
2.6 Partitioning	22
2.7 Z-Ordering	22

2.8	Databricks	23
2.9	Spark Optimization	24
2.9.1	Resource Allocation Management	24
2.9.2	Executor Memory Management	24
2.9.3	Optimization of Spark Shuffling	25
2.9.4	Maximising Parallelism	25
2.9.5	Data Caching	26
2.9.6	Query Optimization	26
2.9.7	Spark Parameter Tuning	27
2.9.8	Data Partitioning	27
2.10	Theoretical Approaches for Column Recommendation	28
3	Proposed Solution	32
3.1	Loading Data From Event Logs	32
3.2	Parsing Relevant Data	32
3.3	Recommendation System and Statistics	33
3.4	Method Illustration	34
4	Implementation	35
4.1	Parsing of Logs	35
4.1.1	Storing of Event Logs	35
4.1.2	Event Logs to Tables	36
4.1.3	Data Transformation	38
4.2	Rule-Based Recommendation	44
4.2.1	Recommendation System's Tables	44
4.2.2	Preprocessing and Data Preparation	45
4.2.3	Simple Count	46
4.2.4	Timestamp Weights	48
4.2.5	Discrete Timestamp Weights	50
4.2.6	Run Time Weights	51
4.3	Statistical Calculations	52
4.4	Pipeline	54
5	Experiments and Results	55
5.1	Method Experiment	56
5.1.1	Query Group 1	56
5.1.2	Query Group 2	57
5.1.3	Query Group 3	58
5.2	Recommendation System Experiment	59

6	Discussion	63
6.1	Experiment Validation	63
6.2	Discussion of Results	64
6.2.1	Method Experiment	64
6.2.2	Recommendation System Experiment	66
6.3	Limitations in Methodology	67
7	Future Work	68
7.1	Enhanced Parsing of the Physical Plan	68
7.2	Method with Multiple Weights	68
7.3	Automation	69
7.4	Multiple Platforms	69
7.5	Dashboard	69
8	Conclusion	70
	Bibliography	75
A		75
A	Supplementary Information	
A.1	Github repository and Dataset	
A.2	Code	
A.3	Figures and Illustrations	
A.4	Tables	

Chapter 1

Introduction

1.1 Motivation

The explosion of big data has led to new opportunities inside business intelligence and data analysis, but it has also caused new challenges for companies to solve. A common approach is migrating to the cloud and storing data in a data lake, allowing for easy access, analysis, and conduction of extract transform and load (ETL) processes. By not storing the data in an optimized way, these processes can be more time-consuming and costly than necessary.

1.2 Problem Definition

When doing ETL and data analysis, companies utilize a myriad of tables. As a result, numerous queries are performed daily against these tables. A common problem is that companies lack information on users' query patterns, making it hard to select important columns to structure the stored data in an optimized way. Even if the company extracts the query pattern information on users for a period, it still tends to change over time, which makes the information obsolete.

A company presented us with the challenge of creating a logging system to optimize the ETL process on their data. Earlier, the data of the company were scattered into different sources on-premise. The company decided to create a new solution, where data from

1.3 Use Case

various sources goes through ETL and is then stored in a common repository in the cloud, namely the Azure Data Lake. The technology utilized by the company to process data into the cloud uses Spark. Most of the data processing and analysis takes place in Azure Databricks and Azure Synapse Analytics. Lastly, the company uses Power BI for the visualization of analyzed data.

1.3 Use Case

The solution proposed in this thesis aims to:

- Lay the foundations for a more structured way of storing information from Spark's event logs.
- Use information from the event logs to give recommendations on which columns to use for partitioning and Z-ordering based on columns; frequently filtered on, recently used for filtering, filter run times.

1.4 Research Questions

1. Can Spark's physical plans be used to optimize partitioning and Z-ordering on tables?
2. Can rule-based column recommendation methods be used to improve the partitioning and Z-ordering of tables?
3. Can the use of continuous updates of partitioning and Z-order be used to optimize queries in an ETL platform?

1.5 Challenges

The main platform utilized in this thesis was Azure Databricks. Azure Databricks offers many different ways to monitor and extract detailed information about the activities on the platform. In the start phase of the research, we wanted to extract the original query string together with the run time. Thus a deep exploration was exercised into the wide number of

1.5 Challenges

logging types found in Databricks and Spark, which included tools like the Query History API, Databricks audit logs, Spark event logs, and Spark Driver logs.

We first investigated the Query History API, which seemed to contain the needed information in an already structured format. However, further research revealed that the monitoring tool was specially designed to track queries performed in DatabricksSQL (a data warehouse). This made us move away from the Query History API as our research was contained in Databricks notebooks, which did not utilize a data warehouse.

Next, we explored Databricks' audit logs. Activating the verbose audit logs made it possible to gain a lot of interesting information from the execution of a cell in a Databricks notebook, such as the command text and duration of the execution. Although this contained much of what was required, it did not satisfy our needs. The reason for this was that multiple Spark queries could be executed in one cell, and even though one has the execution time for the whole cell, there was still no way to extract the run times of the individual queries.

Lastly, we started to look at the event and driver logs. After inspecting the two logs, we discovered that most of the information in the driver logs was based on the event logs. Thus, we directed our focus more toward the event logs. Here, we uncovered events directly connected to the execution of queries. These events gave information about the duration of the queries, but we found no way to obtain the original query string.

After further investigation, there was finally a satisfactory solution. We came across Spark's physical plan, which could be obtained from the query-related events. From the physical plan, we managed to extract the needed query filter information, and as the duration of queries was contained in the events, we finally got the information we needed.

At the start of our work, we planned to study the interaction between Spark and Azure Synapse Analytics. However, because of our research into obtaining the query information described above, this became impossible due to time constraints. We also planned to create visualizations for our recommendations in Power BI, but for the same reasons as described for Azure Synapse Analytics, this did not happen. Another factor was that we had to wait several weeks before getting access to the Microsoft Azure platform and its services, such as Azure Databricks. Having to wait for this seriously slowed down our progress.

Another challenge was the tight security practiced in Bouvet and Equinor in terms of permissions. When performing the research for the query information, we frequently had to request new permissions to find out if services, such as the activation of audit logs and access to log analytics, had the relevant information for our project. The process of waiting

1.6 Outline

for access when wanting to explore new research directions was tiresome and hindered the efficiency of our exploration. Although the tight security is understandable in today's situation, with a more effective working environment, we would have managed to complete several of our ideas, which have been added to the future work section.

1.6 Outline

Chapter 2 - Background:

The background chapter is divided into two parts. First, the needed technical and theoretical background is presented. In the second part, we look at optimization techniques in Spark and introduce our approaches used for column recommendation.

Chapter 3 - Proposed Solution:

This chapter presents the reader with an overview of our proposed solution for the optimization of Spark in an ETL platform.

Chapter 4 - Implementation:

The implementation chapter describes how the different steps were developed in code, from the parsing of data to the calculation of statistics and column recommendations.

Chapter 5 - Experiments and Results:

This chapter introduces the experiments and showcases their results. The first experiment tests the method's abilities, while the second simulates a more realistic use case for the recommendation system.

Chapter 6 - Discussion:

In the Discussion chapter, we examine results from chapter 5 and some of the limitations of the methodology presented in the thesis.

Chapter 7 - Future Work:

This chapter explores some alternative approaches that were not pursued in this thesis. Additionally, we discuss potential technical improvements that could be made in future work to further increase the performance of the implemented methods.

Chapter 8 - Conclusion:

The last chapter concludes the thesis and answers the research questions presented in section 1.4.

Chapter 2

Background

The thesis focuses on Spark optimization. However, it is beneficial to study components interacting with Spark as these are part of the use case we wanted to optimize. This chapter introduces these relevant components and attempts to describe several optimization techniques for Spark. Lastly, we present the theory behind our recommendation methods used in the proposed solution.

2.1 ETL

The ETL process is a vital part of data integration used in the field of data science and big data. It is a three-phase process consisting of extracting, transforming, and loading data. ETL processes gather data from different sources, process it, and store it in a common container. The company's platform uses Azure Databricks to perform ETL processes and uses Azure Data Lake for storage, which is the component we want to optimize. In the first phase, the extraction phase, data sources are identified, and the optimal approach for retrieving the required data is determined. Next, data is transformed to correct errors, handle missing values, and converted into a format fitting the final destination. After transforming the data, it is loaded into the final destination enabling end-users to use it for various purposes such as reporting or analysis. [21, Ch. 1]

2.2 MapReduce

2.2 MapReduce

The MapReduce (MR) pattern is a model for processing large-scale datasets in a distributed manner. Google introduced it in 2004 as a way to process data in a scalable and distributed fashion. The MR model used the map and reduce functions from functional programming as inspiration. During its map phase, the data is transformed into key-value pairs and processed in parallel on different workers. In the reduce phase, the results of the map phase are combined into a final output. [9] [15]

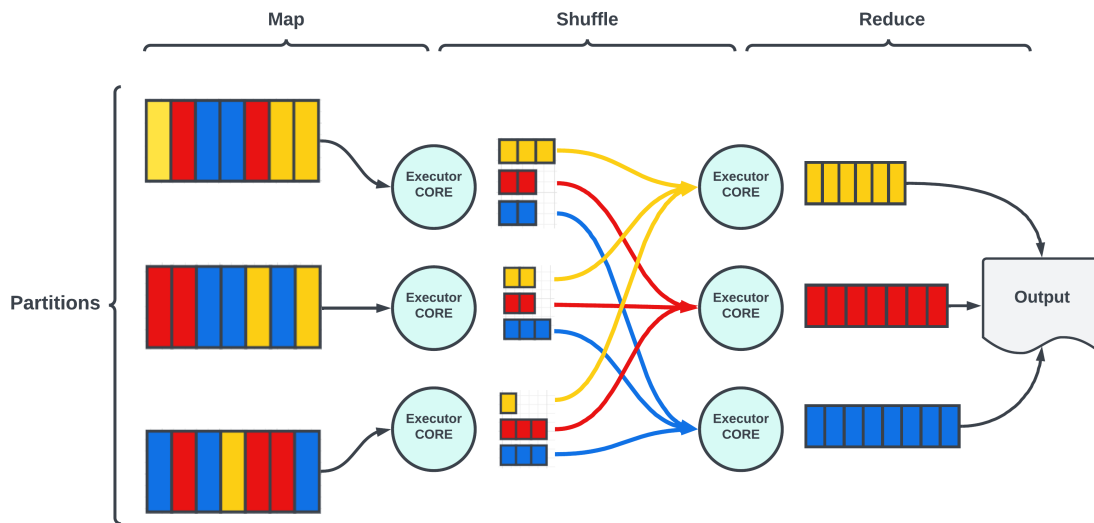


Figure 2.2.1: Illustration of how the MapReduce model works. Each machine in the computation cluster is called an executor. The illustration is based on figure 2. in [27].

A simple run-through of how MR works is shown in figure 2.2.1. Chunks of data (or partitions; explained in 2.6) are run on a cluster of machines with multiple cores. In the **Map** phase, each core in the cluster is assigned partitions to process. The cores read its assigned partitions and map each value it encounters to a key-value pair. In the **Shuffle** phase, the intermediate key-value pairs are transferred between nodes in the cluster to group all pairs with the same key together (see figure 2.2.1). For the **Reduce** phase, each node processes one or more partitions of intermediate data and produces a final result for each unique key. [9] [15]

2.3 Apache Spark

Apache Spark was then released in 2010 as an improvement to the MapReduce model and uses a directed acyclic graph (DAG) to orchestrate computations instead of being limited to the Map, Shuffle, Reduce steps used in the MR pattern [6] [36] [5, p. 4].

2.3 Apache Spark

Apache Spark is a framework for distributed data processing, and it stores data in-memory to increase processing speed. Another aspect is that Spark has APIs in multiple programming languages, such as Python, Java, Scala, and R. As mentioned, Spark takes care of processing, but the task of permanent storage of data is given to storage systems such as Azure's data lake, Amazon S3 or Hadoop.

Two common approaches for processing data are batch and streaming, and Apache Spark has functionality for both. After the data is processed, one can use Spark SQL to query the data for further analysis or even machine learning (ML). MLlib is Spark's ML library, which contains algorithms commonly used for ML. Lastly, the GraphX library can be used for visualizing the results from analysis or ML. [30]

2.3.1 Architecture

Spark is a framework where workloads are distributed to different worker nodes for computation. A worker node is responsible for starting executors. An executor is a Java virtual machine (JVM) that is used for computation and storage. A collection of worker nodes containing executors are referred to as a cluster. To orchestrate the worker nodes and executors in the cluster, there is a need for components to manage communication. In the next part, we will talk about five important components at the higher level in Spark as described in "Learning Spark" [5, pp. 10-12]. The components are visible in an illustration which can be seen in figure 2.3.1. [5, pp. 10-13]

2.3 Apache Spark

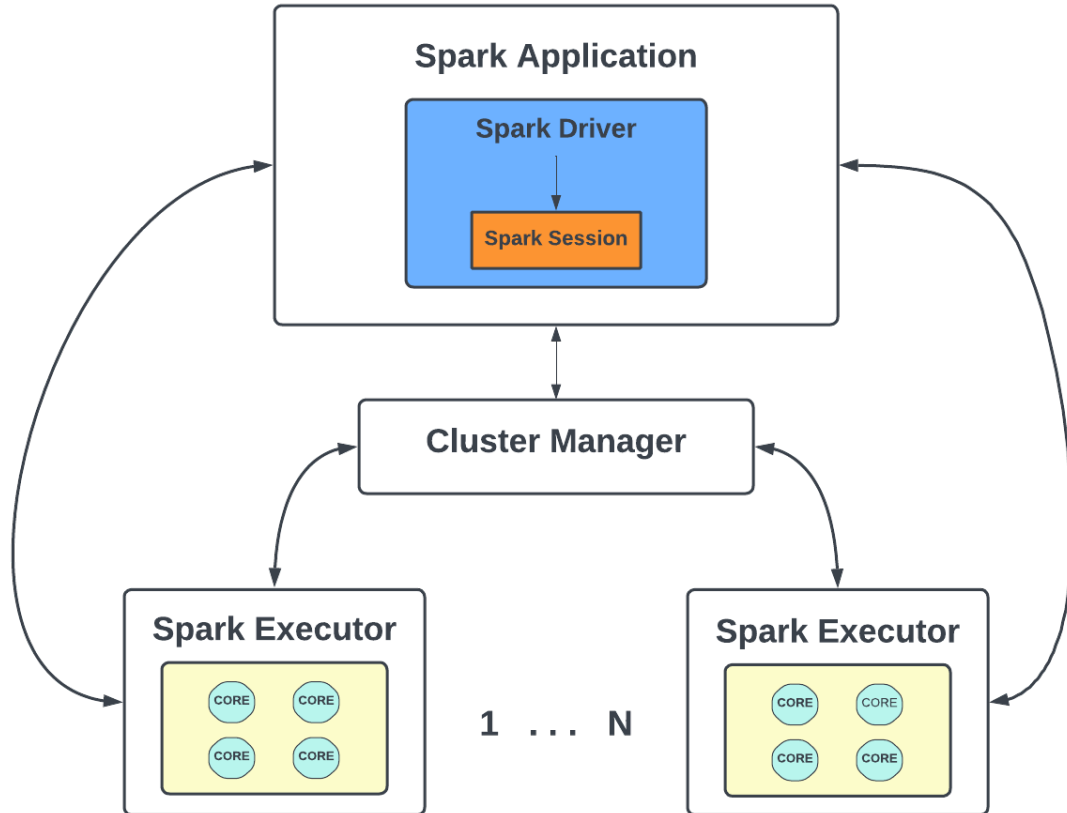


Figure 2.3.1: Sparks architecture containing the five components: Spark Application, Spark Driver, Spark Session, Cluster Manager, and the Spark Executor (Illustration is based on figure 1-4 from [5, p. 10]).

- The **Spark Application** is simply the outer shell encompassing the Spark Driver component and the Spark Session.
- The **Spark Driver** has a few different applications. It is responsible for starting the Spark session and requests resources from the Cluster Manager to execute operations. After the resources have been distributed, there is no need for communication with the cluster manager and the Spark driver will talk directly to the executors. This communication consists of distributing the tasks between the executors so that one can achieve parallel processing. In addition, it also manages the transformation from

2.3 Apache Spark

Spark operations to directed acyclic graphs, which are then handed out as tasks.

- **Spark Session** is a new conduit that has merged multiple entry points such as SparkContext, SQLContext, HiveContext, and more. The creation of the session makes it possible to utilize all the functions of Spark. Unifying all the entry points allows one to perform multiple different operations, such as creating DataFrames, datasets, reading data, and creating JVM parameters, along with others.
- As mentioned earlier, the **Cluster Manager** allocates resources for the Spark Driver. It communicates with the cluster and requests the needed computation power from the nodes to carry out the jobs. A few different cluster managers are available, like Apache Hadoop YARN, the built-in standalone cluster manager, Kubernetes, and Apache Mesos.
- The last component is the **Spark Executor**. Each of the worker nodes in a cluster has one or multiple Spark executors. The Spark executor executes tasks assigned to it and communicates with the Spark driver.

2.3.2 APIs in Spark

Running workloads on the Spark cluster requires the usage of Spark SQL or either of its APIs. To understand why some APIs in Spark are preferred when it comes to the optimization of Spark, we will take a look at Spark's three main APIs, namely the **RDD**, **Dataframe**, and the **Dataset**. [5, pp. 43-69, 84]

2.3.2.1 Resilient Distributed Dataset

The RDD, or Resilient Distributed Dataset, is an essential concept in Spark and is used as a building block for the structured higher-level APIs, namely DataFrames and Datasets. The RDD has three main properties; dependencies, partitions, and a compute function. RDDs use a list of *dependencies* instructing Spark which inputs and preceding RDDs are needed to recreate the RDD, which is why they are called "Resilient". *Partitions* divide the data into smaller chunks that can be processed in parallel by the Spark cluster. While performing computations on a set of partitions, Spark tries to assign the partitions to nodes as close as possible to the partition's physical location. Doing so reduces the need for data movement between nodes. The *compute function* is part of the RDD responsible for performing computation on the data. Having processed the data, it returns an iterator

2.3 Apache Spark

over the function's output. This iterator is not a typed variable, meaning it can return any data type, depending on the data being processed. The function's generic return type limits Spark's ability to "understand" the data, thus hindering how much optimization can be done by solely using the RDD programming API. [5, p. 44] [10]

Even though using the RDD programming API directly makes it hard for Spark to optimize the executions, RDDs still serve as the backbone for the higher-level APIs discussed in the following paragraphs. When using the higher-level APIs, data transformations are ultimately transformed into RDDs, but now in a way that lets Spark optimize the computations. The computations are translated into a DAG where each node represents an RDD, and the edges represent transformations performed to the RDD. The optimization process and how higher-level APIs are turned into RDDs are explained in section 2.3.4. [5, pp. 4, 6, 44-47] [20, Ch. 3]

2.3.2.2 DataFrame

A Spark **Dataframe** is a distributed collection of data partitions presented as an abstract data object with named columns. The DataFrame is conceptually the same as a relational database table but optimized for distributed processing. DataFrames can be created from multiple data sources such as structured data files, tables in Hive, external databases, or existing RDDs [32]. When the DataFrame is initialized with some data, it can be transformed and manipulated using a set of built-in functions or by using the *df.sql()* syntax to execute a SQL query. [5, pp. 47-68]

We mentioned earlier how the opaque iterator returned from the RDD's compute function limited the amount of optimization Spark was able to do. This is where RDDs and DataFrames differ; DataFrames use a schema to define the data types for each column, whereas RDDs do not. Using this schema helps Spark to understand the data better, qualifying Spark to create an optimized execution plan for the transformations. Spark DataFrames also use a more expressive programming syntax than RDDs letting the users specify the transformations in a more expressive and abstract way. To showcase the difference in expressiveness, we have borrowed two code examples from "Learning Spark" [5, p. 45-46] show in listings 2.1 and 2.2. [5, pp. 45-68]

2.3 Apache Spark

```
1 dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
2 ("TD", 35), ("Brooke", 25)])
3 # Use map and reduceByKey transformations with their lambda
4 # expressions to aggregate and then compute average
5 agesRDD = (dataRDD
6 .map(lambda x: (x[0], (x[1], 1)))
7 .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
8 .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

Listing 2.1: Code example of how one would find the average number of values belonging to each key by the use of RDD.

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import avg
3 # Create a DataFrame using SparkSession
4 spark = (SparkSession
5 .builder
6 .appName("AuthorsAges")
7 .getOrCreate())
8 # Create a DataFrame
9 data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ...
10 ("Jules", 30),
11 ("TD", 35), ("Brooke", 25)], ["name", "age"])
12 # Group the same names together, aggregate their ages, and compute an ...
13 average
14 avg_df = data_df.groupBy("name").agg(avg("age"))
```

Listing 2.2: This code is producing the same output as the listing above, but is now using the DataFrame API instead of RDD.

2.3.2.3 Dataset

As seen in figure 2.3.2, both the DataFrame and Dataset are unified as structured APIs with the main difference being type checking [5, p. 69]. Compared to the DataFrames, the Dataset API gives better type checking and will notify type errors at compile-time, often better than the counterpart, where errors are raised at runtime [5, p. 75].

Datasets are not available for PySpark as Python is a dynamically typed language, unlike Java and Scala, which are statically typed. In statically typed languages, variables are known at compile-time, whereas in dynamically typed languages, the type of variables is determined at runtime. Having variables determined at runtime makes enforcing type

2.3 Apache Spark

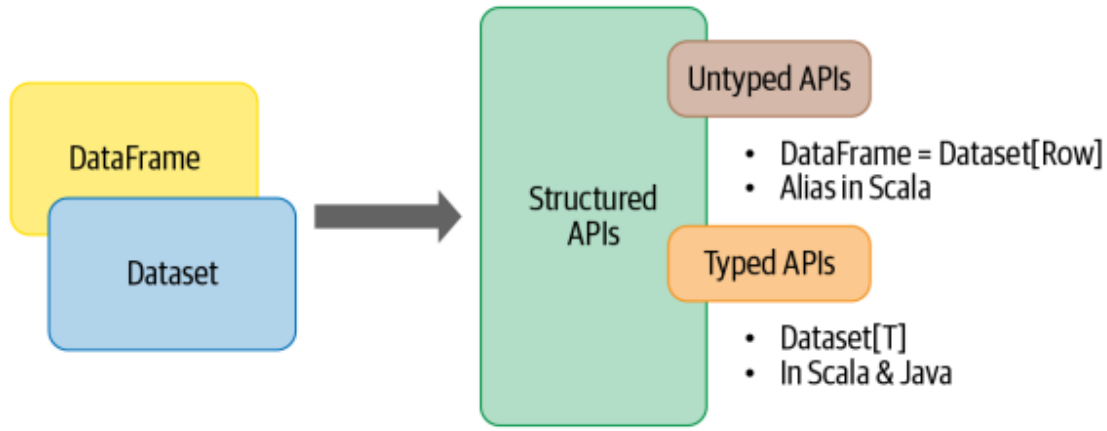


Figure 2.3.2: Structured APIs in Spark. Figure 3-1 from [5, p. 69].

safety in Python difficult, which is a fundamental aspect of the Dataset API. Datasets allow developers to use a Java class as a type for the data inside a DataFrame to manipulate it as a collection of typed objects [8]. Typing aside, both structured APIs allow for better optimization for the transformations happening under the hood, which we will look closer at in section 2.3.4. [5, pp 69-75]

2.3.2.4 Transformations and Actions

It is important to note that Spark's APIs are immutable, meaning they cannot be modified. There are instead created a new object between each transformation. When managing the data in the Spark data abstractions, two main operations are used to interact with the data: *transformations* and *actions*. **Transformations** are the operators that transform the data, which due to the immutability, forces the creation of a new data object [11]. Some of the more common transformations are: *orderBy()*, *groupBy()*, *filter()*, *select()*, *join()*. **Actions** on the other hand, are the operator that "calls for action" meaning that Spark has to start executing the scheduled transformation(s). Spark does not perform any computation until an action operator is run. This is what we call **Lazy-evaluation**; no transformation is executed before it is absolutely necessary (i.e., an action operator is called upon). Up until the point of execution, the logical plan of execution is created and optimized, which we will read more about in section 2.3.4. [5, pp. 28-30]

2.3 Apache Spark

2.3.3 Physical Plan

A crucial part of our solution is parsing information from Spark's physical plan. The data gained from the operations used on tables and columns lays the foundation for how our system recommends columns. That is why we will look closely at Spark's physical plan in the next part.

The Spark physical plan is a low-level plan which describes the plan of execution in detail. It is the last step in the optimization procedure before the query is translated into RDDs, as seen in figure 2.3.4. One can find a tree-like structure in the upper part of the physical plan. The tree contains operation names and a reference number to connect the operation to the more detailed part of the physical plan below. One can obtain column and table names, partition information, and more information in the detailed part, as seen in figure 2.3.3, which shows an example of a Spark physical plan. In the following paragraphs, we will present some of the operations from Spark's physical plans used in our solution, with a basis in the information found in the article [34] by Vrba.

```
== Physical Plan ==
* Filter (3)
+- * ColumnarToRow (2)
   +- Scan parquet spark_catalog.default.tt12 (1)

(1) Scan parquet spark_catalog.default.tt12
Output [5]: [PersonID#2894, LastName#2895, FirstName#2896, Address#2897, City#2898]
Batched: true
Location: PreparedDeltaFileIndex [dbfs:/user/hive/warehouse/tt12]
PushedFilters: [IsNotNull(PersonID), EqualTo(PersonID,2)]
ReadSchema: struct<PersonID:int,LastName:string,FirstName:string,Address:string,City:string>

(2) ColumnarToRow [codegen id : 1]
Input [5]: [PersonID#2894, LastName#2895, FirstName#2896, Address#2897, City#2898]

(3) Filter [codegen id : 1]
Input [5]: [PersonID#2894, LastName#2895, FirstName#2896, Address#2897, City#2898]
Condition : (isnotnull(PersonID#2894) AND (PersonID#2894 = 2))
```

Figure 2.3.3: Spark physical plan example.

The first operation we will talk about is called 'scan parquet'. The 'scan parquet' operation represents the reading of data from parquet files. Here one can attain information such as

2.3 Apache Spark

which database has been used, the table name, and the name of columns from the table. 'scan parquet' uses two different filters, namely 'PartitionFilters', and 'PushedFilters'. 'PartitionFilters' are utilized on a column when the table is partitioned on it. Secondly, we have 'PushedFilters', this filter is used when filtering on columns and pushes filter operations down to the parquet level. What is significant about both these filters is that they are used to skip unwanted data so that one can do less computation and use fewer resources.

The 'Filter' operation contains conditions for columns to be filtered and uses Spark's internal column id to reference the columns. What is important to note, and will be reviewed more thoroughly in chapter 2.3.4 is that Spark performs optimization, which can lead to modified and relocated filters, so they may be slightly changed from their original form before converting them to their physical counterparts.

2.3.4 The Catalyst Optimizer

As mentioned in the section about APIs, using the Spark SQL or either of Spark's structured APIs is needed when we want Spark to "understand" the objective of the code and help optimize the logical steps. What is meant by Spark "understanding" the code's objective is that it will use the **Catalyst** to optimize the execution plan that is to be run; this process is called query optimization. In this subsection, we will look at how the Catalyst performs query optimization and the steps Spark will go through, from the first step of reading the code to the last step, where the optimized code is executed. The information in this section and its subsections are based on chapter three from the book "Mastering Apache Spark" [20, Ch. 3] and from the book "Learning Spark" [5, pp. 77-81].

While performing transformations on any of Spark's data types, one creates and changes the logical plan describing how Spark will handle the data to create the desired output. Once an *action* operator is called on a structured API, the logical plan will be sent to the Catalyst for optimization. Next, we want to introduce the first of four stages in the Catalyst, **Analysis**.

2.3.4.1 Analysis

When an action parameter is called, and the logical plan is sent to the Catalyst, the plan is transformed into an abstract syntax tree (AST). An AST is a tree-like structure

2.3 Apache Spark

representing the transformations included in the logical plan. The tree's leaf nodes are literal values (either constants or values read from data), while the other nodes represent transformations that are to be done on its child nodes. The AST is referred to as the *Unresolved Logical Plan*. Because the AST is created directly from the user's code, Spark does not yet know if the tables, columns, and attributes contained in the logical plan exist.

To resolve these properties, the Catalyst uses an internal Spark data structure called a *Catalog*. The Catalog is an object that contains information about different columns, data types, functions, tables, databases, and more [5, p. 81]. After resolving the plan, a (resolved) logical plan is outputted to Catalyst's next phase, Logical Optimization.

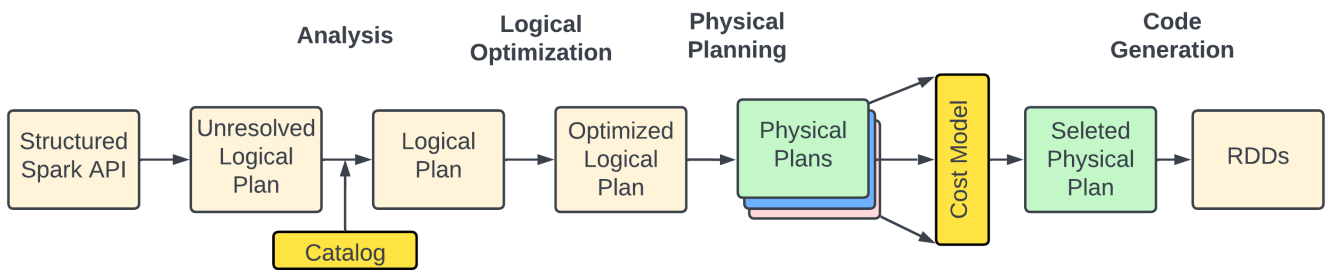


Figure 2.3.4: Catalyst Optimizer. Based on Figure 3-4 from [5, p. 78].

2.3.4.2 Logical Optimization

The **Logical Optimization** is the second phase of the Catalyst. The main goal of this phase is to create a more efficient version of the logical plan that the Spark engine can later run. During this phase, an array of rule-based optimizations are performed to the logical plan. When using these rule-based optimizations, multiple potential "best" plans are created and later evaluated by a cost-based optimizer to assign a cost to each of the plans. The plan with the lowest cost is chosen and sent to the next phase.

2.3 Apache Spark

2.3.4.3 Physical Planning

In this phase, the goal is to generate an optimized physical plan based on the optimal logical plan from the logical optimization. Physical planning is the last phase before the code generation and execution of the plan. It translates the logical plan into a physical plan specifying where to read data and, in detail, how to execute the different operations planned in the logical plan.

As seen in figure 2.3.4, multiple physical plans are created in the physical planning phase. If run, each of these would have returned the same data result, but we are only interested in the most optimized plan. Multiple physical plans are generated, each optimized with different combinations of operators. There are used strategies such as basic operator selection and join strategy selection to decide which physical operators to use. For example, the operator selection chooses the physical operators that will be used to replace the operators from the logical plan, such as filters, sorts, and maps. The join selection helps select the optimal joins resulting in fewer data transfers between the nodes. The code responsible for the physical planning contains different strategies accountable for their own part of the conversion from a logical to a physical plan. [31]

After generating multiple options, heuristics are applied to select the most suitable plan to minimize execution time. This physical plan includes details about which methods to use when accessing data, the join algorithms to use if there are any joins, which operators to use when filtering data, and in which order the operations should be executed (An example physical plan is seen in figure 2.3.3). This plan is then passed to the final phase: code generation.

2.3.4.4 Code Generation

In this last phase of query optimization, **Code Generation**, executable code is generated for each of the operations based on the optimized physical plan. The code generated is in JVM bytecode format and is optimized for the specific query and data processing task. Motivation for the code generation phase is based on Spark transformations often being executed on data located in memory, making the transformations a CPU-bound process. Bytecode optimized for in-memory computations is generated to lessen the CPU-bound bottleneck and optimize the computations. [5, pp. 4, 81]

Code creation uses a property of the Scala programming language, Quasiquotes. Abstract

2.3 Apache Spark

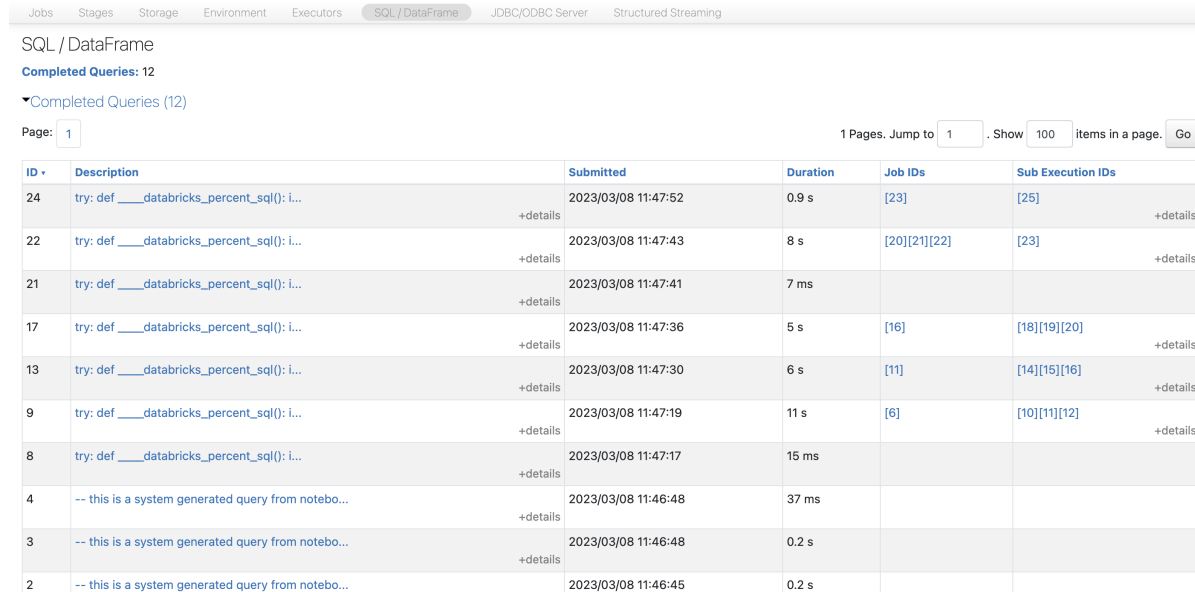
syntax trees created using Quasiquotes can be further processed by the Scala compiler at runtime to produce bytecode. Project Tungsten is an initiative in Apache Spark that aims to increase the performance and efficiency of the execution engine. Using **Project Tungsten**, Spark can also perform whole-stage code generation. [12]

Whole-stage code generation is a technique used to combine different stages of the physical plan when generating code for it. Some of the advantages of combining stages before code generation are that there will be created fewer virtual functions calls (no need for a function per stage), and there will be less data passed through the network when moving between stages as the variables to be used for the next stage is already in the CPU-registers/memory [13] [12].

2.3.5 Spark UI

The Spark user interface (UI) is a tool used for monitoring Spark. In the Spark UI, one can observe how Spark processes data by dividing it into jobs, stages, and tasks. One can also see metrics on the amount of memory the RDD uses and the RDD's size. Detailed information about the Spark environment and the executors is also available in the Spark UI. The most relevant part of the Spark UI for the thesis is the SQL/DataFrame tab, which contains information about user queries that can also be found in the Spark event logs. A snippet from the SQL/DataFrame tab in the UI can be seen in figure 2.3.5 the rows represent Spark queries. [5, p. 31]

2.3 Apache Spark



The screenshot shows the Spark UI interface for the SQL/DataFrame tab. At the top, there are navigation tabs for Jobs, Stages, Storage, Environment, Executors, SQL/DataFrame (selected), JDBC/ODBC Server, and Structured Streaming. Below the tabs, it says "SQL/DataFrame" and "Completed Queries: 12". There is a dropdown menu for "Completed Queries (12)" and a pagination control showing "Page: 1" and "1 Pages. Jump to 1, Show 100 items in a page, Go". The main content is a table with the following columns: ID, Description, Submitted, Duration, Job IDs, and Sub Execution IDs. The table contains 12 rows of query execution data.

ID	Description	Submitted	Duration	Job IDs	Sub Execution IDs
24	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:52	0.9 s	[23]	[25] +details
22	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:43	8 s	[20][21][22]	[23] +details
21	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:41	7 ms		
17	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:36	5 s	[16]	[18][19][20] +details
13	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:30	6 s	[11]	[14][15][16] +details
9	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:19	11 s	[6]	[10][11][12] +details
8	try: def ___databricks_percent_sql(): i... +details	2023/03/08 11:47:17	15 ms		
4	-- this is a system generated query from notebo... +details	2023/03/08 11:46:48	37 ms		
3	-- this is a system generated query from notebo... +details	2023/03/08 11:46:48	0.2 s		
2	-- this is a system generated query from notebo... +details	2023/03/08 11:46:45	0.2 s		

Figure 2.3.5: Spark UI SQL/DataFrame tab.

2.3.6 Spark Event Logs

Spark event logs are Spark's way of storing information about the computation performed. In the execution process, Spark emits Spark events, which are dictionary-like objects containing keys and values that are saved to the Spark event logs. In the execution of a stack of code, a vast amount of Spark events are produced. Most Spark events have unique types of keys specific to the event, but there also exist keys that can be the same for multiple events. One such key is the 'Event' key, which exists in all Spark events and is used to identify the type of event. The following section will mention the events encountered in our proposed solution.

The first Spark event we will discuss has the identification string "org.apache.spark.sql.execution.ui.SparkListenerSQLExecutionStart". This event is emitted each time Spark executes a SQL query and indicates the start of a query. It has the 'Event' key to indicate the type of event with an identification string, the 'executionId'; a query execution id, the 'rootExecutionId'; the root execution id of a query, 'description'; a string snippet from the executed code, 'details'; a key with strings containing execution details, the 'physicalPlanDescription' containing a detailed plan of which operations Spark will utilize when

2.3 Apache Spark

executing on the physical level, the 'sparkPlanInfo' that has a tree-like structure built on information from the physical plans operations, 'time'; holding a Unix time integer with information about the start time of the query, and lastly, the 'modifiedConfigs' key; including details about the changed configuration are located. Figure 2.3.6 shows an example of the Spark event emitted at the start of query execution.

```
{
  "Event": "org.apache.spark.sql.execution.ui.SparkListenerSQLExecutionStart",
  "executionId": 5,
  "rootExecutionId": 6,
  "description": "df.show(1)",
  "details": "org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecu",
  "physicalPlanDescription": "== Physical Plan ==\nLocalTableScan (1)\n\n\n(1) Local",
  "sparkPlanInfo": {
    "nodeName": "LocalTableScan",
    "simpleString": "LocalTableScan [default.date#1511L, version_time#211L, dir#202",
    "children": [
      ],
    "metadata": {
      },
    "metrics": [
      {
        "name": "number of output rows",
        "accumulatorId": 700,
        "metricType": "sum",
        "experimental": false
      }
    ],
    "explainId": "None"
  },
  "time": 168129102345,
  "modifiedConfigs": {
    "spark.r.sql.derby.temp.dir": "/tmp/RtmpRPmeUH",
    "spark.databricks.optimizer.optimizeLimit.maxLimit": "1001"
  }
}
```

Figure 2.3.6: Shows a snippet example of a Spark event log.

The second Spark event we will look at has the identification string "org.apache.spark.sql.ex

2.4 Apache Parquet

ecution.ui.SparkListenerSQLExecutionEnd". This event occurs at the end of a the execution of a query, containing many of the same keys as the Spark event emitted at the execution start. Three of its four keys are the same as in the query start event namely, 'Event', 'executionId', and 'time'. The last key is called 'errorMessage' and stores execution errors.

2.4 Apache Parquet

Apache Parquet is a file format where data is stored in a hybrid (combination of column-based and row-based) way [4], compared to the traditional row form. Apache Parquet uses data skipping to reduce the number of I/O operations. In the following paragraphs, we will explain why a hybrid approach is taken and how Parquet performs data skipping.

With the information presented in the lecture [4] by Braams, we will look at the workings of the Apache Parquet format. The first step is introducing the terms Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP is a type of workload where several small operations are performed on entire rows. On the other hand, OLAP is associated with analytical operations, such as performing aggregates on columns. There are typically three ways to store data in files: row-based, column-based, and hybrid-based. Parquet uses the hybrid type. We introduced OLTP and OLAP because row-based is effective on OLTP workloads but not so much for OLAP workloads and vice versa for column-based. Thus, we got the hybrid which takes advantage of both the row-based and the column-based storing types. For a better understanding of the methods used for storing data, an illustration was created, which is seen in figure 2.4.1.

2.5 Azure Data Lake

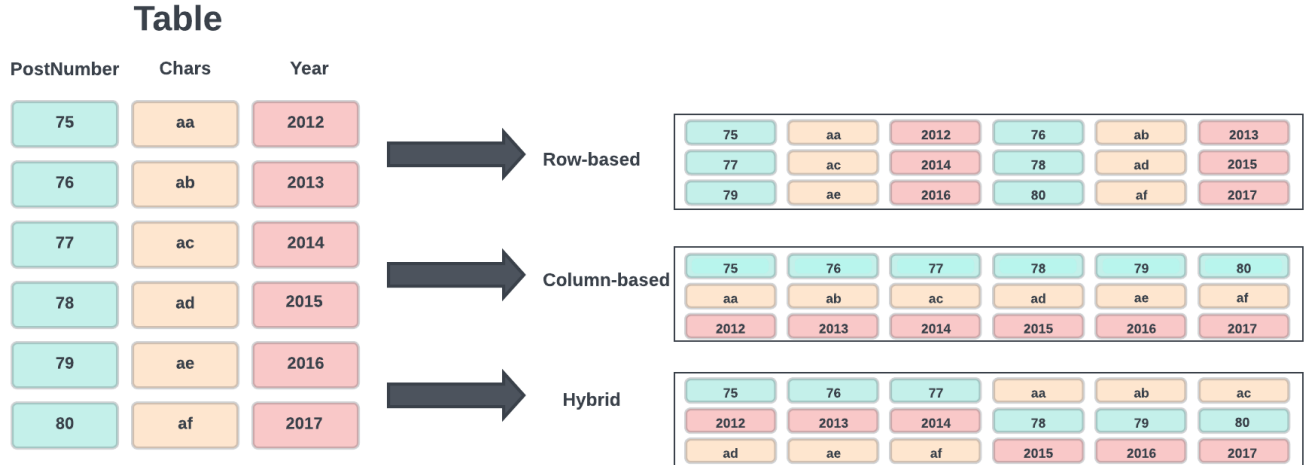


Figure 2.4.1: Illustration of the most common ways to store data in files (Illustration was based on figure 2 from [2]).

Another aspect of the Parquet format is that it stores metadata for each column chunk. If we again look at the illustration 2.4.1 and the hybrid one, we can see that there are chunks of three values from each column and that this pattern repeats until we have all the values from the table. Parquet stores metadata for each column chunk making it possible to do data skipping. [23]

Lastly, Parquet is the standard format used in delta tables, which is needed to perform Z-ordering. Thus, having a general understanding of the Parquet file format is essential for the rest of the thesis.

2.5 Azure Data Lake

The Azure Data Lake is a storage service in the Microsoft Azure cloud. The Data Lake allows storing of structured, semi-structured, and unstructured data. As a result, ingested data can be stored in its raw format. The Data Lake is scalable so that its data storage can be changed as the size of the data changes. Azure Databricks, which will be discussed later, is closely integrated with the Azure Data Lake, making for easy access to data. [1]

2.6 Partitioning

The Azure Data Lake can also be extended by the [Delta Lake](#), which brings atomicity, consistency, isolation, and durability (ACID). The Delta Lake supports the creation of Delta tables and has the Z-ordering (2.7) functionality for optimization. [17, Ch.1]

2.6 Partitioning

In figure 2.2.1, we gave an example of how partitions in a distributed storage are used together with transformations in a MapReduce framework. The Apache Parquet file format was introduced in section 2.4, where we presented how metadata is stored for each file partition. The next paragraphs will give an analogy of how partitioning works on a file level so that one can better understand how data structuring can lead to more effective data processing.

Let us start with a simple analogy; picture a library with thousands of books grouped by the author's surname. Each bookshelf in the library represents a partition, and the label on the bookshelf corresponds to the metadata for that partition. The first bookshelf might contain books written by authors starting with a specific surname, say 'name A', while the second bookshelf might have books written by authors with the surname 'name B'. If we translate this analogy to the digital world, we could say that our data (in this case book data) is stored in Parquet files which are partitioned on the column "Author's surname".

Imagine querying book data with a SQL WHERE clause, including the author's surname. The first step would be to find the bookshelf containing the author's surname and only look inside that single bookshelf. If the data were un-partitioned, one would have to go through the whole library, book by book. To summarize, data partitioning is merely a way of structuring the stored data to optimize the retrieval of data. Doing so on a commonly used column can help reduce the number of files that need to be read from disk, thus optimizing the time and computation power needed to perform tasks on an ETL platform.

2.7 Z-Ordering

Partitioning works best on columns with low cardinality. It would not make sense to run partitioning on an id-column with all distinct values; this would create a "bookshelf" for each data point, only creating more overhead when reading the data. Luckily, according to Databricks documentation for Z-ordering, optimization on a high cardinality column is

2.8 Databricks

where the Z-ordering works best [7].

A Z-order curve is a mathematical way of mapping multi-dimensional data to a single dimension. By using the "ZORDER" command the Delta lake will utilize a Z-order curve to keep related data points in the same file [7]. Z-ordering can be performed on a combination of columns in a delta table, but it can also be applied to a single column [7]. In the Partition section, we introduced an analogy of how each bookshelf was a partition including books from authors with the same surnames. Let us visualize a similar scenario where people visiting the library, were more interested in the book's title and see how this can be applied with the use of Z-ordering.

The books could be partitioned by their titles. However, this would lead to a vast number of partitions, one for each unique title, most likely a single partition per book, and one would quickly lose the benefit of having partitions in the first place. A book title column is likely a property with high cardinality, which implies that Z-order could be beneficial [7]. If "Book Title" was chosen as the property to structure data on, using Z-ordering would result in a sorting of all the books by looking at their titles (dimensionality reduction) and inserting the sorted books into the bookshelves. By recording the metadata from the bookshelves (translates to metadata in parquet files), one can label each bookshelf with its first and last book title following alphabetical order, e.g., shelf1: ("Alice in Wonderland", "Cat's Cradle") implying books with titles in this range are stored in this bookshelf ¹.

Looking at the change of interest in book properties from the author's surname to the book's title is similar to how users' query patterns change on tables. It demonstrates the advantage of having a system to read which columns are used for data filtering and to be able to recommend important columns that should be used to optimize the data storage.

2.8 Databricks

Databricks is a Lakehouse platform where it is possible to perform Data Warehousing, Data Engineering, Data Streaming, and Data Science plus Machine Learning. The original creators of Spark created Databricks to have a platform integrated with the cloud where one could utilize Spark [5, p. 34]. Databricks can be used with different cloud providers, but in this thesis, we will focus on Azure Databricks since this will be used.

¹Note the difference between the bookshelves in the partition analogy compared to the Z-order analogy. When Z-ordering; each bookshelf is a file containing a range of values, using partitions, each bookshelf was a folder, e.g., "surname=Jacobsen".

2.9 Spark Optimization

Databricks are supporters of open source, and as a result, technologies such as Delta Lake, MLflow, Apache Spark, and more have been further developed and integrated into the Azure Databricks platform. One of these technologies is the Workflows, which is Databricks' own tool to build pipelines where one can extract, load, and transform data. It makes it possible to utilize a combination of notebooks, SQL queries, Spark, and so on for the ETL process [14]. The Workflows also provide a monitoring system where one can spot faults.

2.9 Spark Optimization

There is a myriad of ways to improve the efficiency of Spark. Most of them, in one form or another, include tuning Spark parameters, but other options exist. The optimization techniques will be presented in a way where we explain them in their simplest formats and then look at how people have expanded on these techniques or even developed new ones. Chapter seven from "Learning Spark" [5, Ch. 7] presents some of these optimization techniques and will be used as a base when explaining the techniques in their simpler formats in the following subsections.

2.9.1 Resource Allocation Management

The first optimization technique we will look into is resource allocation. Resource allocation is related to how Spark distributes its resources, such as the number of executors utilized or the amount of resources given to a single executor. Allocating too many resources for a single query execution can lead to reduced performance on other executions running in parallel. The paper by Sen et al. [29] tackles this problem by creating a predictive system called the AutoExecutor, which predicts run times based on the number of allocated executors. The idea is that by having predictive data on query run times, users can, in advance, choose a more optimal configuration for the number of executors.

2.9.2 Executor Memory Management

Another aspect is how the memory of the executors is distributed in Spark. The executor memory is divided into three main parts, that is the execution memory, storage memory, and the reserve memory. The shuffling of data is one of the heaviest operations performed

2.9 Spark Optimization

in Spark, which includes reading and writing from the disk. Spark performs shuffles using the execution memory, which means that by changing the size of the three executor memory parts, one can allocate the right amount of memory needed for the different types of queries to reduce I/O operations. The ratio can be set by the **Spark.memory.fraction** property which controls how much memory is used for execution and storage. To further manipulate the fraction between execution and storage, the **Spark.memory.storageFraction** can be used.

2.9.3 Optimization of Spark Shuffling

As mentioned earlier, optimizing Spark's shuffling is a great way to increase efficiency. That is why Spark allows the user to affect how the operation is carried out. By taking advantage of properties such as **spark.shuffle.file.buffer** one can help minimize the number of times I/O operations occur before the shuffle partitions are finally written to the disk. It is also possible to tweak the shuffle service. A drawback of the shuffle service is that its index files are opened multiple times. Facebook proposed a solution to this problem [19], instead of reopening the file each time, the data is instead stored in a [least recently used](#) (LRU) cache. The size of the LRU cache can be set by using the **Spark.shuffle.service.index.cache.size** property.

2.9.4 Maximising Parallelism

When doing data computation in parallel, the goal is to evenly distribute the data into partitions for the different executors, such that one gets maximum parallelism. To do this, Spark, already at the file level, read data as partitions [5, p. 181]. Data skew, which refers to uneven partition sizes, is a common problem related to the parallelization process [3]. Paul et al. [24] proposed a new system called Optimizing Data Partitioning for In-Memory Data Analytics Frameworks (CHOPPER) to handle Spark's in-memory data skew problem. CHOPPER can automatically choose the best number of partitions for workloads and change how data is divided and distributed during the execution of a workload. As a result, better workload handling was achieved, improved parallelism, and reduced shuffle traffic. The paper by Huang et al. [18] also tries to solve or at least reduce the impact of data skew. Using linear regression, they try to predict the size of partitions before moving to the reduce stage. They use the predicted partition sizes to distribute the data evenly between the executors in the reduce stage.

2.9 Spark Optimization

2.9.5 Data Caching

In the next part, we will look at caching as an optimization technique. In Spark, we have both `cache()` and `persist()` which have many similarities, but there is a key difference; when using `persist`, one can specify where to store the data, while using `cache` one cannot. Applying a cache can considerably enhance performance when DataFrames or tables are used multiple times, such as in ETL processes. This is because when the cache operator is invoked, it will store as many of the object's partitions as possible in-memory, reducing the number of I/O operations needed.

2.9.6 Query Optimization

A common way to optimize Spark is through query optimization. One approach to query optimization is to influence Spark's choice of join, as these utilize heavy operations. There are five join types in Spark, and picking the right one based on the data size is essential to reduce the number of shuffling operations performed by Spark. Li et al. [22] introduce a query optimizer called Runtime Integrated Optimizer for Spark (RIOS), which influences Spark's join process in the following way; it dynamically picks the join order, modifies the execution plan, and can base its choice of join in the physical plan on data gathered from runtime statistics. An aspect that sets their work aside from others is that they have created an optimizer, which also works on User Defined Functions (UDFs).

Another costly part related to queries is stateful operators. Stateful operators maintain some state of the data, meaning the operator's output depends not only on the current data point but also on the maintained state from previously processed data [28, Ch. 18]. This is costly as information is exchanged over the network. The paper by Modi et al. challenges the problem of stateful operators and tries to reduce the cost by using multiple query optimization techniques. The optimization techniques include a new state-of-the-art algorithm for the exchange of data, which resulted in a reduction of transferred data, three partial push-downs that obtain supporting operators from the main operators, and then push the supporting operators further down the execution tree. Lastly, they use a technique called 'peephole optimizations' to improve the implementation of the stateful operators.

2.9 Spark Optimization

2.9.7 Spark Parameter Tuning

Configuring Spark with the proper parameters in regard to how data will be processed on the Spark system is a crucial task. Picking the wrong parameters for Spark can be costly and give inefficient run times. As a result, different teams have proposed solutions for auto-tuning parameters on Spark SQL applications. Xin et al. [35] made a system called Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications (LOCAT) for this purpose. LOCAT uses Bayesian optimization, giving the system low overhead compared to other Machine Learning techniques. In the paper "You Only Run Once: Spark Auto-Tuning from a Single Run" [25], Prats et al. introduce an auto-tuner that tunes the Spark parameters by using metrics extracted from a single run of a Spark application. They build a feature descriptor that parses the Spark event log files generated when performing the initial run. This process makes a feature vector that contains a summary of the Spark tasks and stages performed by the workload. They start with a standard Bayesian Optimization process and build upon it using transfer learning from the execution logs. The transfer learning from the execution logs allows the model to better generalize unseen workloads without retraining, which is a big advantage as they claim the state-of-the-art work based on performance model search techniques has the drawback that retraining is required to generalize unseen workloads. Prats et al. prove that using a "Simulated Bayesian Optimization" (SBO) approach drastically (12x) decreases the "time to solution" compared to the standard Bayesian Optimization approach. The solution also achieved up to an 80% increase in performance compared to the standard Spark configuration [25].

2.9.8 Data Partitioning

The last optimization technique we will go into, which is also the category of our solution, is how structuring data into partitions can help optimize Spark. Searching for the data matching a query can be costly if one has to go through the whole data set. The research by Guo et al. [16] proposes a solution with a system called Adaptive Partitioning Scheme for Exploratory Queries (APEQ), which uses self-created plugins on top of Spark to gather data-driven and user-driven metrics used in a partition learning engine to learn the best partitions. They also use a partition index tree containing metadata about partitions and a Query Optimization Engine that utilizes the index tree to search for partitions. Their system generates multiple partitions, which are directed at common queries. Instead of focusing on the importance of columns for partitioning as we do in our approach, they instead create individual partitions for common filters from queries, which for example, can be a specific filter spanning ten years on a column for years. When queries are executed,

2.10 Theoretical Approaches for Column Recommendation

their approach checks the index tree for matching partitions. If a viable partition is found, then they will change where the query accesses the data from the non-partitioned table to the single partition, which fits the requirements of the query.

Guo et al., before Spark execution, send the user queries directly to a metric plugin library to collect user-driven metrics on the queries, which are later used to learn optimal data partitions. As opposed to their approach, we collect data from Spark's event logs after execution. By using Spark's event logs after execution, we are able to use parameters such as query run time as a parameter for our recommendation system, which based on our interpretation of their system, they do not.

2.10 Theoretical Approaches for Column Recommendation

In our practical work we decided to focus on partitioning and Z-ordering of tables by the use of Spark event logs. To make the following sections easier to understand, we will explain some background of the **approaches** we chose to pursue. We define approaches as the idea and mathematics behind each of our implemented recommendation **methods**. In contrast, by methods, we refer to the concrete implementations of the recommendation methods. It is important to note that the metrics in the following subchapters are calculated on a "per database-table-column" basis and are afterward ordered in a descending order where the result at the top is chosen as the recommended column to partition the data on.

2.10.0.1 Simple Count

The first approach is a count of the times each column is used as a filter on a table. The rationale behind this approach is that columns commonly used for filtering are the ones that the tables should be partitioned on.

Let O be a record of all the operations/filters performed (found in the logs). O_{C_i} is a subset of O containing all the operations filtering on the column C_i . F is the filter used for the filter operation, F also includes metadata about the filtering, i.e., information about when the filtering took place; F_t . We can define the simple count metric (SC_{C_i}) as the sum of the number of times each column is used as a filter on a table, subject to the condition $X(F)$.

2.10 Theoretical Approaches for Column Recommendation

$$SC_{C_i} = \sum_{F \in O_{C_i}} X(F) \quad (2.1)$$

where $X(F)$ is a window function (with input parameters t_{start} and t_{end} in the format of Unix timestamp):

$$X(F) = \begin{cases} 1 & \text{if } t_{start} \leq F_t \leq t_{end} \\ 0 & \text{otherwise} \end{cases} \quad (2.2)$$

The function $X(F)$ is defined such that it evaluates to 1 if the time F_t falls within the specified time window and 0 otherwise. This function filters out operations that do not fall within the desired time range and only considers those that do.

2.10.0.2 Timestamp Weights

The previous approach was founded on the idea that frequently filtered columns are more likely to be filtered on again. The rationale for this approach is similar but emphasizes the recency of the queries; recent queries are more similar to future queries and, therefore, should be given an appropriate weight before performing the aggregation of operations to calculate a reflecting score.

We bring the window function (equation 2.2) from the previous method but add a function, $T(F)$ to calculate the appropriate weights based on the recency of the queries:

$$T(F) = \begin{cases} w_{max} & \text{if } F_t \geq t_{end} \\ \frac{(F_t - t_{start})}{(t_{end} - t_{start})} \times (w_{max} - w_{min}) + w_{min} & \text{if } t_{start} < F_t < t_{end} \\ w_{min} & \text{if } F_t \leq t_{start} \end{cases} \quad (2.3)$$

It uses the same input parameters as the window function $X(F)$: t_{start} and t_{end} , but it also has weights associated with the start and end of the period marked by the window parameters. The function $T(F)$ uses [linear interpolation](#) to calculate the appropriate weight on filters taking place inside the interval. Approaching the end of the interval (t_{end}) will give you a weight closer to w_{max} , which will increase the impact for the current filter. Vice

2.10 Theoretical Approaches for Column Recommendation

versa for towards the start (t_{start}) of the interval. Applying this function to the summation we had earlier¹, we get:

$$TW_{C_i} = \sum_{F \in O_{C_i}} X(F) \cdot T(F) \quad (2.4)$$

2.10.0.3 Run Time Weights

This last approach uses the run time of the physical plan from where the filter originates. The idea is that more complex operations with longer run times should have a larger impact on the method's recommended column compared to the less complex queries with shorter run times.

An important thing to note here is that not all filters that appear in complex physical plans with a long run time may be the reason for the long run time. However, the idea is that aggregation over multiple physical plans will capture the columns reappearing in complex physical plans with extended durations. These columns are likely being used in heavy operations and should be used for Z-ordering and partitioning of tables.

The function denoted by $R(F)$ calculates the appropriate weight based on the filter's physical plan's run time (F_r). The first Input parameter to the function, $R(F)$, is r_{max} . This parameter is the run time needed to reach the maximum weight, w_{max} , being the second input parameter. The last parameter, w_{min} , dictates the weight when the run time is 0s. Alike the previous approach (2.10.0.2), operations having durations in between the interval between 0 and r_{max} seconds will get an appropriate weight between the interval of the weights w_{min} and w_{max} .

$$R(F) = \begin{cases} w_{max} & \text{if } F_r \geq r_{max} \\ \frac{F_r}{r_{max}} \times (w_{max} - w_{min}) + w_{min} & \text{if } 0 < F_r < r_{max} \\ w_{min} & \text{if } F_r \leq 0 \end{cases} \quad (2.5)$$

¹It is redundant to have both the window function 2.2 as well as the timestamp weight function 2.3 as the latter also handles data points being outside of the interval. It is nevertheless kept, as the original window function is part of the preprocessing happening before the calculation of weights

2.10 Theoretical Approaches for Column Recommendation

Applying the above function to the Simple Count equation 2.1, we get an approach that takes run time into consideration:

$$RTW_{C_i} = \sum_{F \in O_{C_i}} X(F) \cdot R(F) \quad (2.6)$$

Chapter 3

Proposed Solution

We propose an optimization technique for Spark, where one takes advantage of the information available in Spark's event logs. We read events containing information about queries to extract their time of execution, run times, and the filter operations used from Spark's physical plans. The information gathered is used to create a recommendation system for what columns to use for partitioning and Z-ordering of tables. The recommendation system is based on four rule-based methods, each looking for different query characteristics.

3.1 Loading Data From Event Logs

The event logs produced by Spark contain much information, but this information is scattered into multiple files. To make the information more accessible for data processing, we store the data in a table.

3.2 Parsing Relevant Data

After storing the data in a more structured format, the next step is parsing the relevant information. To do this, we first have to manipulate the data to transform it into a proper JSON format. Next, the JSON objects are exploded and expanded into a table where the

3.3 Recommendation System and Statistics

keys are represented as columns and events as rows. The events representing queries are extracted from the newly created event table, and yet another table for queries is built.

From the query event table, parsing is performed on the column 'physicalPlanDescription', which is a column that contains the physical plan for a specific query. Information about databases, tables, columns, and different types of filters are extracted by the use of regex, and this is stored in a final table (A.3.2) later used as input for the recommendation system and statistical calculations.

3.3 Recommendation System and Statistics

Using information from Spark's physical plan and event logs, we recommend the optimal columns to partition and Z-order on. We propose four different recommendation methods based on the three approaches² discussed in section 2.10.

The first method uses the 'Simple Count' approach to count over the filters used on each column inside a specific time interval. The second and third methods use the 'Timestamp Weights' approach. The second method calculates weights based on how recent the filter operations are. The third method divides the time interval into n equally sized partitions containing filter operations. These partitions are then manually given a weight based on their importance. The fourth and last method uses the 'Run Time Weights' approach, which uses the run time of the physical plan to calculate a weight to give higher weights to filter operations with high run times.

We divide data into intervals of n-weeks for statistical calculations. The timestamp of queries is then used to calculate different statistics for individual columns and tables, such as aggregates.

²Note method 2 and 3 uses the same approach, namely 'Timestamp Weights', this is how we end up with three approaches and four methods.

3.4 Method Illustration

3.4 Method Illustration

In the figure 3.4.1 one can see an illustration of the proposed solution.

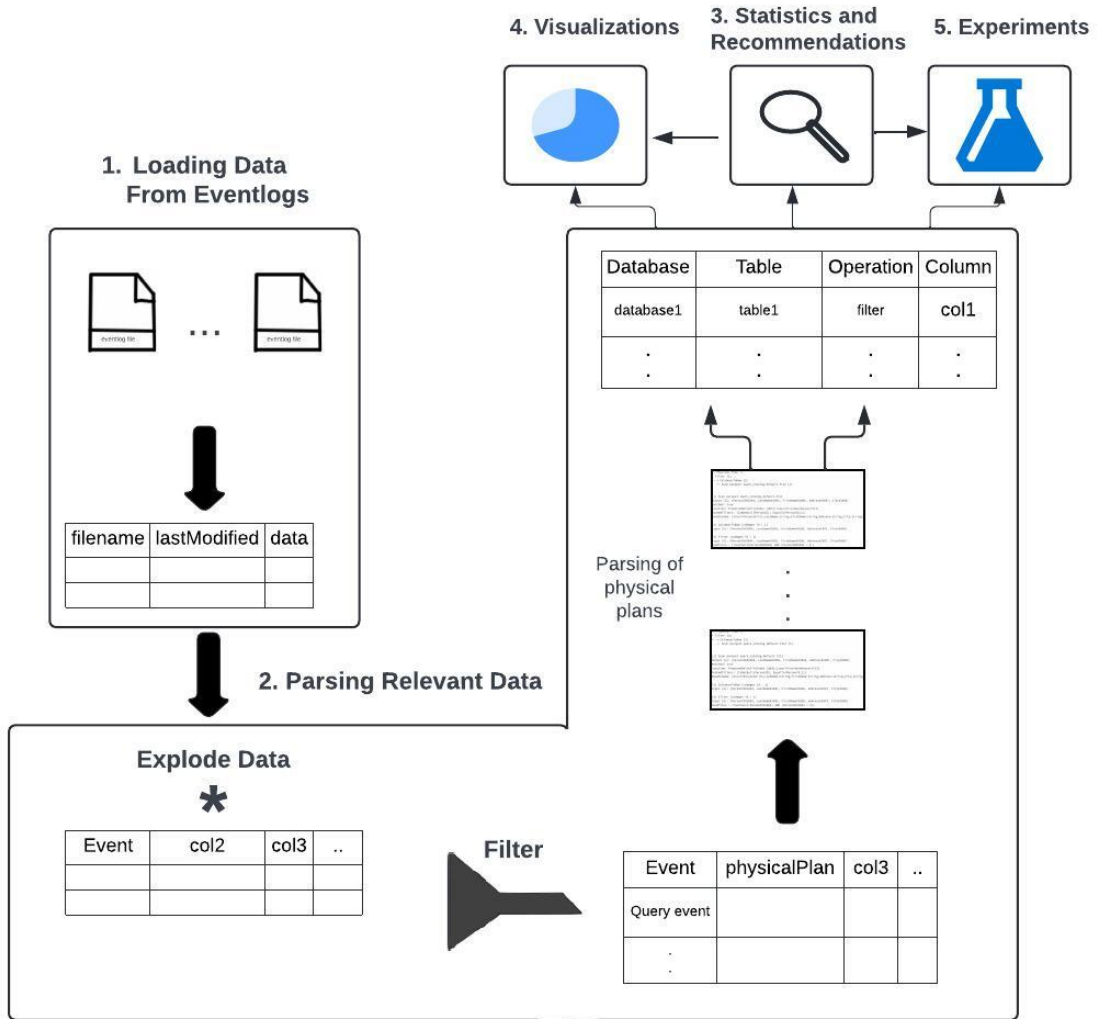


Figure 3.4.1: Illustration of the proposed solution.

Chapter 4

Implementation

This chapter presents a detailed description of the steps taken in the implementation of our column recommendation system; extracting information from event logs, implementing recommendation methods, and calculating statistics. The work has been performed using Python, PySpark, and Databricks. Figure A.3.1 shows the relation between the tables used in this chapter.

4.1 Parsing of Logs

This section describes how we extracted relevant information from Spark's event logs. More specifically, the `SQLExecutionStart` and `SQLExecutionEnd` events and the `'Filter'`, `'PushedFilters'`, and `'PartitionFilters'` operations which are all explained in the section about Spark's physical plans (2.3.6).

4.1.1 Storing of Event Logs

Azure Databricks uses Spark clusters for computations. However, the event logs are not stored as default. Without further configuration, the event logs are only available when the cluster is active and are deleted when the cluster is terminated. To counter this, we maneuvered into the advanced cluster options and chose a destination for the storage of

4.1 Parsing of Logs

event logs.

▼ Advanced options

Azure Data Lake Storage credential passthrough ⓘ

Enable credential passthrough for user-level data access

Spark Logging Init Scripts JDBC/ODBC Permissions

Destination ⓘ

✔ Last delivered at 2023-04 CEST

Figure 4.1.1: Image of what the configuration of storing the Spark event logs look like in Databricks.

4.1.2 Event Logs to Tables

The next step is to read the files, capture the relevant metadata and store them in a table. The notebook responsible for these tasks is divided into two main parts. The first part reads the event logs and gathers available file metadata. The final part reads this data into a Spark DataFrame (DF) and appends the data to the `eventlog_raw` table.

In the first part, we start by traversing through all the files in the log directory, gathering the file metadata. The contents of the metadata can be seen in the variables in line three in listing 4.1. The event log file's timestamp is found by either using the modification timestamp from Python's Pathlib [26] library or by extracting the timestamp from the filename. The other properties are found from the file's path, which is chosen by Spark on storage. The metadata and the file contents for each file are linked inside a common tuple and stored in a list. Next, we use the notebook's input parameters `start_time` and `end_time` to find files in the relevant time interval. The updated list (variable `relevant_files` in listing 4.1) is used to read the file data of each relevant file.

³Finished event logs are zipped by Spark, and given a timestamp to the filename, while unfinished are not.

4.1 Parsing of Logs

Listing 4.1: Loading data from files. Code from eventLogToDelta.ipynb.

```
1 data = []
2 for file_meta in relevant_files:
3     path_object, cluster_id, cluster_instance_id, spark_context_id, ...
4         last_modification_timestamp, is_zipped = file_meta
5
6     file_data = ''
7     if is_zipped:
8         with gzip.open(str(path_object), 'rb') as g:
9             file_data = g.read().decode('UTF-8')
10    else:
11        with open(str(path_object), 'r') as f:
12            file_data = f.read()
13
14    row = (str(path_object), cluster_id, cluster_instance_id, ...
15          spark_context_id, last_modification_timestamp, file_data)
16    data.append(row)
```

In the final part of the notebook, we use the list of tuples and a Spark schema to create a DF. We construct a unique event log key from three of the values in the DF: 'clusterInstanceID', 'sparkContextID', and 'lastModified'. By using this unique key, we can be sure that we do not have any duplicate event logs in our tables. The *event_logs* DF is finally written to the *eventlog_raw* table, ready for transformations.

Listing 4.2: Constructing a data frame from the tuples in the data list. Code from eventLogToDelta.ipynb.

```
schema = StructType(
    [
        StructField("filePath", StringType(), True),
        StructField("clusterID", StringType(), True),
        StructField("clusterInstanceID", StringType(), True),
        StructField("sparkContextID", StringType(), True),
        StructField("lastModified", TimestampType(), True),
        StructField("fileData", StringType(), True),
    ]
)
event_logs = spark.createDataFrame(data=data, schema=schema)
```

4.1 Parsing of Logs

4.1.3 Data Transformation

This section will describe the transformations used to extract query and filter information from the event logs.

We started by transforming the event log data to a proper JSON string. The event logs stored events inside curly braces separated by newline characters. (i.e., {event1}\n{event2}\n{...}), we needed to transfer them to a JSON-formatted list of objects.

We accessed event log data from the column 'fileData' in the eventlog_raw table. Pyspark's regex function was then used to remove the newline characters and afterward replaced the occurrences of "}" with "},{", to separate each event with a comma. We then added square brackets to the start and end of the string, transforming it into a JSON list. The next step was reading the JSON list with Pyspark's from_json function; we needed a schema containing all of the relevant keys to be found in the JSON string. The schema (seen in appendix, A.1) was used to transform the JSON string to a list of JSON objects. The transformations described in this paragraph are shown in listing 4.3.

Listing 4.3: Transforming the event log data to MapType. The schema variable is shown in the appendix A.1. Code from RawDataToOrganized.ipynb.

```
import pyspark.sql.functions as F

df = spark.sql("SELECT * FROM eventlog_raw")
df = df.withColumn("fileData", F.regexp_replace("fileData", "\n", ""))
df = df.withColumn("fileData", F.regexp_replace("fileData", "\\}\\{", "},{"))
df = df.withColumn("fileData", F.concat(F.lit "["), df.fileData, ...
    F.lit "]" ))
df = df.withColumn("fileData", F.from_json("fileData", schema))
```

After the previous step, we have a DF where each row represents an event log in the form of a list of event objects. By executing an [explosion](#) on the list of objects, we give each of the events its own row in the DF, seen as *df2* in listing 4.4. Further, to make each of the distinct event keys into columns, we do an expansion using the "columnName.*" syntax seen in the last line of the same listing. The result is a DF, where columns are event keys and rows are events.

4.1 Parsing of Logs

Listing 4.4: Exploding and expanding the data into separate columns. Code from RawDataToOrganized.ipynb.

```
columns = [
    "filePath",
    "clusterID",
    "clusterInstanceID",
    "sparkContextID",
    "lastModified",
    "eventlogKey",
]
df2 = df.select(*columns, F.explode(df.fileData).alias("eventData"))
df3 = df2.select(*columns, F.col("eventData.*"))
```

Using the expanded DF, we can now move on to the next step. Filtering for the 'SQLExecutionStart' and 'SQLExecutionEnd' events which represent the start and the end of a query executed in Spark. To gather all relevant information about a query, we want to join the information found in both the SQL start and end events. We use *df3* as a base, create a DF for each of the SQL events, and join the DFs on the query's unique ID columns. Code snippets⁴ for this step can be seen in listing 4.5 and 4.6. With the new DF, the queries' run times are calculated based on the timestamps of the start and end events.

Listing 4.5: Filtering to create DataFrame for the SQLExecutionStart events. Code from RawDataToOrganized.ipynb.

```
sql_start_df = df3.select(
    df3.Event.alias("event_start"),
    "sparkContextID",
    "clusterInstanceID",
    "executionId",
    "rootExecutionId",
    "Description",
    "Details",
    "physicalPlanDescription",
    "sparkPlanInfo",
    df3.time.alias("time_start"),
    "modifiedConfigs",
    "eventlogKey",
).filter(
    df3.Event == "org.apache.spark.sql.execution.ui.SparkListener
    SQLExecutionStart"
)
```

⁴To prevent presenting the same code multiple times, we only show how we create the `sql_start_df`, not the `sql_end_df` as it uses the same logic.

4.1 Parsing of Logs

Listing 4.6: Join and calculation of duration. Code from `RawDataToOrganized.ipynb`.

```
sql_df = sql_start_df.join(
    sql_end_df,
    (sql_start_df.executionId == sql_end_df.executionId)
    & (sql_start_df.clusterInstanceID == sql_end_df.clusterInstanceID)
    & (sql_start_df.sparkContextID == sql_end_df.sparkContextID),
).drop(sql_end_df.executionId, sql_end_df.clusterInstanceID, ...
       sql_end_df.sparkContextID)

sql_df = sql_df.withColumn(
    "duration_ms", sql_df.time_end.cast("double") - ...
                  sql_df.time_start.cast("double")
)
```

Each row in the DF now represents a single SQL execution, including its physical plan. As discussed in background section 2.3.3 about physical plans, we want to specifically look at the occurrences of the filter operations: 'Filter', 'PushedFilters', 'PartitionFilters', and the 'scan parquet' operation. By looking directly at the filter operations from the physical plan, one can only see the column names with an internal id, e.g., "columnName#42". Columns in different tables can have the same names, and because of this, we had to find a way of differentiating the columns and linking them to their appropriate table. We solve this problem by looking at the 'scan parquet' operation, which tells us which database and table the column belongs to. We made the function `create_column_lookup` parsing the physical plan and returning a column lookup dictionary. This function can be seen in listing 4.7.

Listing 4.7: Shows the `create_column_lookup` function as well as its integration into a UDF. Code from `RawDataToOrganized.ipynb`.

```
def create_column_lookup(physicalPlan):
    COLUMN_LOOKUP = {}

    regex = r"(\d+)\s+Scan parquet\s+(\S+)\nOutput \[\d+\]: \[(.*?)\]"
    matches = re.findall(regex, physicalPlan)

    for m in matches:
        db_table = m[0]
        database, table = db_table.split(".")[-2:]

        columns = m[1]
        for c in columns.split(", "):
            COLUMN_LOOKUP[c] = {"database": database, "table": table}
    return COLUMN_LOOKUP
```

4.1 Parsing of Logs

We built three functions to parse the physical plans, which are all closely connected. The first is 'build_rows' seen in listing 4.8. The function takes in results from the parsing of a single physical plan with a single filter operation e.g., 'PartitionFilters'. Its input parameters are: 'columns'; all the columns used for this filter, 'column_lookup'; the output from the function explained above, and some parameters containing shared metadata of the physical plan. The function's primary purpose is to create a list of rows for each column used in a specific filter type and is used inside the plan_parser, explained next in listing 4.9.

Listing 4.8: Shows the 'build_rows' function. Code from RawDataToOrganized.ipynb.

```
def build_rows(
    columns,
    column_lookup,
    operation_name,
    execution_id,
    timestamp,
    physical_plan_key,
    eventlog_key,
    table=None,
    database=None,
):
    unique_columns = set(columns)
    rows = []
    for c in unique_columns:
        if database is None:
            database = column_lookup.get(c, {}).get("database", "")
        if table is None:
            table = column_lookup.get(c, {}).get("table", "")

        rows.append(
            {
                "operationName": operation_name,
                "executionId": execution_id,
                "databaseName": database,
                "tableName": table,
                "columnName": c.split("#")[0],
                "timeGenerated": timestamp,
                "physicalPlanKey": physical_plan_key,
                "eventlogKey": eventlog_key,
            }
        )
    return rows
```

4.1 Parsing of Logs

The second function is responsible for parsing physical plans and focuses on one operation at a time. It is named *plan_parser* and takes in the following parameters:

- **row** the row in the DF.
- **name** the name of the filter operation.
- **regex** regex searching for the filter operation.
- **col_regex** regex for extracting the column names from the operation regex's output.
- **group_index** integer for specifying which capture group in the regex that contains the column names.

For each of the operations found, we use the column regex to extract the names of all columns related to the operation filter. The *build_rows* function is called and returns a data row for each column inputted. Finally, after the loop has finished, the *row_lst* variable is returned.

Listing 4.9: Shows the 'plan_parser' function. Code from RawDataToOrganized.ipynb.

```
def plan_parser(row, name, regex, col_regex, group_index):
    execution_id = row["executionId"]
    timestamp = row["timestamp"]
    physical_plan = row["physicalPlanDescription"]
    column_lookup = row["columnLookup"]
    physical_plan_key = row["physicalPlanKey"]
    eventlog_key = row["eventlogKey"]

    row_lst = []
    matches = re.findall(regex, physical_plan)
    for m in matches:
        cond_group = m[group_index]
        columns = re.findall(col_regex, cond_group)

        if name == "PushedFilters": # Missing internal id so needs a ...
            small tweak
            db_table = m[0]
            database, table = db_table.split(".")[2:]

        rows = build_rows(
            columns,
            column_lookup,
            name,
```

4.1 Parsing of Logs

```
        execution_id,
        timestamp,
        physical_plan_key,
        eventlog_key,
        table=table,
        database=database,
    )
else:
    rows = build_rows(
        columns,
        column_lookup,
        name,
        execution_id,
        timestamp,
        physical_plan_key,
        eventlog_key,
    )
    row_lst.extend(rows)

return row_lst
```

Last, we have the *multi_plan_parser* function, as seen in listing 4.10. To avoid looping through the whole DF multiple times, the 'multi_plan_parser' was created. The function evaluates a single physical plan at a time while looping through the filter operations specified in the *settings* variable seen in listing 4.11. The two input parameters are the *settings* variable specifying which filter operations to look for together with corresponding regex expressions, and a row from a DF needed for the plan_parser function call.

Listing 4.10: Shows the 'multi_plan_parser' function. Code from RawDataToOrganized.ipynb.

```
def multi_plan_parser(row, settings):
    row_instances = []
    for operation in settings:
        operation_rows = plan_parser(
            row,
            operation["operation_name"],
            operation["regex"],
            operation["col_regex"],
            operation["group_index"],
        )
        row_instances.extend(operation_rows)
    return row_instances
```

4.2 Rule-Based Recommendation

Listing 4.11: Shows how the 'multi_plan_parser' was called together with settings for the "Filter" operation. Code from RawDataToOrganized.ipynb.

```
settings = [
  {
    "operation_name": "Filter",
    "regex": "\\(\\d+\\)\\s+Filter(?:\\n)Input(?:\\n)Condition : ...
      (.*?)\\n\\n",
    "col_regex": "\\w+#\\d+",
    "group_index": -1,
  },
  ...
]

operation_rdd = physical_plan_lookup_df.rdd.map(
  lambda row: multi_plan_parser(row, settings)
)
operation_df = operation_rdd.flatMap(lambda l: l).toDF()
```

In the last step of this notebook, the operations and queries tables are created. Each row in the operations table is a filter operation extracted from a physical plan, and each row in the queries table is a combined SQLExecutionStart and SQLExecutionEnd event aggregation. The tables and columns used for storage can be seen in the entity diagram A.3.1.

4.2 Rule-Based Recommendation

Now that we have the operations and queries tables, we have the needed prerequisites to create a recommendation system for partitioning and Z-ordering on columns. We used four different methods utilizing the three approaches explained in section 3.3 and 2.10. Each of these four methods has its own notebook with some overlap for the data preparation as a natural first step.

4.2.1 Recommendation System's Tables

The tables: method_runs, method_results (A.3.3), and method_recommendations (A.3.4) are used to store the recommendation methods' metadata, results, and column recommendations, respectively. The method_runs table contains metadata for each time a recommendation system is run and a 'runId' that is used as a foreign key in the two other tables.

4.2 Rule-Based Recommendation

Resulting evaluations of each column are found in the `method_results` table. These evaluations are used to build the `method_recommendations` table, which is the final table outputting the columns recommended by each method. For more details on column names and table relations, look at the entity diagram in figure A.3.1.

4.2.2 Preprocessing and Data Preparation

The notebooks' inputs are validated and set to the correct format, and we perform three steps of shared preprocessing:

1. Remove data points outside the window of interest marked by the notebook's input parameters `to_time` and `from_time`. These parameters translate to the window function presented in equation 2.2.
2. Remove operations with the filter type: 'Filter'⁵.
3. Remove any missing values, either in the form of an empty string or a NULL value.

Listing 4.12: Preprocessing performed in all notebooks as a initial step before performing any calculations on the operations table. Code from `RuleBased_count.ipynb`.

```
operations = (  
    spark.sql("SELECT * FROM operations")  
    .filter(F.col("timeGenerated").between(from_time_timestamp, ...  
        to_time_timestamp))  
    .filter(F.col("operationName") != "Filter")  
)  
  
operations = operations.select(  
    [  
        F.when(operations[col] == "", ...  
            None).otherwise(operations[col]).alias(col)  
        for col in operations.columns  
    ]  
)  
)  
.dropna(how="any")
```

⁵We decided to do this since we noticed that the 'PushedFilters' and 'Filter' were always present in pairs. And for our implemented methods, this would have led to a doubling in importance for these filters compared to the 'PartitionFilters'.

4.2 Rule-Based Recommendation

Listing 4.13: Schema used to read a metadata dictionary object into `method_runs` table. Code from `RuleBased_count.ipynb`.

```
method_runs_schema = StructType(  
    [  
        StructField("runId", IntegerType(), nullable=True),  
        StructField("methodName", StringType(), nullable=True),  
        StructField("params", StringType(), nullable=True),  
        StructField("fromTime", TimestampType(), nullable=True),  
        StructField("toTime", TimestampType(), nullable=True),  
        StructField("whenRun", TimestampType(), nullable=True),  
    ]  
)
```

Before doing recommendation calculations, we want to see which columns the tables are partitioned on. Using the operations table, we find the newest filter occurrence of every column and check if the operation is of type 'PartitionFilters'. We use this information to fill the `is_partitioned` column in the `method_result` and recommendation tables. Having gone through these common steps of the recommendation methods, we can start by looking at the implementation details of the different methods.

4.2.3 Simple Count

The names associated with the Simple Count's metadata parameters give a concise explanation of what they do, so we deem it unnecessary to provide further explanation. Note that the *interval* parameter used as part of the *params* dictionary (listing 4.14) is a parameter that tells the window size in terms of weeks and is represented as a float.

Listing 4.14: Parameters and metadata to be written to the `method_runs` table (4.13). Code from `RuleBased_count.ipynb`.

```
params = {  
    "windowStart": int(from_time.timestamp() * 1000),  
    "windowEnd": int(to_time.timestamp() * 1000),  
    "windowSize": interval,  
}  
method_run_info = {  
    "runId": runId,  
    "methodName": "simpleCount",  
    "params": json.dumps(params),  
    "fromTime": from_time,  
    "toTime": to_time,  
    "whenRun": datetime.now(),  
}
```

4.2 Rule-Based Recommendation

```
}
method_run = spark.createDataFrame([method_run_info], ...
    schema=method_runs_schema)
```

We use a similar logic to the approach explained in section 2.10.0.1 by executing a `groupBy` operation that counts the number of occurrences of each column present inside the relevant window in the operations table (A.3.2). In listing 4.15, you can see how the method is executed using the `groupBy` operator. The output of this step gives us the `method_results` table (A.3.3). The `method_results` table has a column 'methodValue' representing the number occurrences for each column.

Listing 4.15: Simple Count: calculating the number of times each table is present in the operations table. Code from `RuleBased_count.ipynb`.

```
method_results = (
    operations.groupBy("databaseName", "tableName", "columnName")
    .agg(F.count("executionId").alias("occurrences"))
)
# Include information on column partitioning
method_results = method_results.join(
    is_partitioned, on=["databaseName", "tableName", "columnName"]
)
```

To find the method's recommended columns, we have to perform a `groupBy` operation on each database and table combination and perform an aggregation to find the column name of the highest-scoring column.

In order to keep the values of the columns 'methodValue', 'columnName', and 'isPartitioned', we need to map these three values into a struct data type before applying Sparks' `max()` function in the aggregation. This struct is then split into three columns matching the schema of the `method_recommendations` table being the final destination.

Code for the latter part is shown in listing 4.16 and will produce an output equal to the one shown in the `method_recommendations` table shown in the appendix (A.3.4).

4.2 Rule-Based Recommendation

Listing 4.16: Calculating the number of times each table is present in the operations table. Code from RuleBased_count.ipynb.

```
method_recommendations = (  
    method_results.groupBy("databaseName", "tableName")  
    .agg(  
        F.max(F.struct("methodValue", "columnName", ...  
            "isPartitioned")).alias(  
                "max_methodValue_colName_isPartitioned"  
            )  
    )  
    .select(  
        "databaseName",  
        "tableName",  
        "max_methodValue_colName_isPartitioned.columnName",  
        "max_methodValue_colName_isPartitioned.methodValue",  
        "max_methodValue_colName_isPartitioned.isPartitioned",  
    )  
    .withColumn("runId", F.lit(runId))  
    .withColumnRenamed("max_methodValue_colName_isPartitioned.columnName", ...  
        "columnName")  
    .withColumnRenamed(  
        "max_methodValue_colName_isPartitioned.methodValue", "methodValue"  
    )  
    .withColumnRenamed(  
        "max_methodValue_colName_isPartitioned.isPartitioned", ...  
        "isPartitioned"  
    )  
)
```

The sections for the following methods will be shorter as there is no need to explain similar parts of the code. Instead, We will focus on the implementation of the calculation of weights and the difference in parameters.

4.2.4 Timestamp Weights

The Timestamp Weights method is built upon the previously mentioned principles in the background section 2.10.0.2. Compared to the Simple Count method, this has some additional input parameters. The parameters: *min* and *max_weights* are used to calculate the appropriate weights for each row of the operations table before further computations.

Listing 4.17: Parameters belonging to the timestamp weights method. Code from Rule-Based_weightedCount.ipynb.

4.2 Rule-Based Recommendation

```
params = {
    "max_weight": max_weight,
    "min_weight": min_weight,
    "windowStart": from_time_timestamp,
    "windowEnd": to_time_timestamp,
    "windowSize": interval,
}
```

Listing 4.18 shows how the weights are calculated based on the 'timeGenerated' column in the operations table. The *interval_weight* variable in this listing uses a logical expression to check if the timeGenerated value is before or after the window of interest marked by the to and from timestamp variables. If the timestamp is located before or after the window of interest, this current row will receive the min or max weights, respectively. It is important to note that the operations DF is already constructed with the start and end time, so no data should be outside the window of interest. Thus the expression in the *interval_weight* variable serves as an additional check to ensure that the weights are correctly assigned to the data points within the window. However, should the timeGenerated value be located inside the window, we use linear interpolation to calculate the appropriate weight.

Listing 4.18: Code showing how the weight is calculated based on the timeGenerated column. Code from RuleBased_weightedCount.ipynb.

```
interval_weight = F.when(
    F.col("timeGenerated") ≥ to_time_timestamp, max_weight)
    .otherwise(
        F.when(F.col("timeGenerated") ≤ from_time_timestamp, min_weight)
        .otherwise(
            (
                (F.col("timeGenerated").cast("long") - from_time_timestamp)
                / (to_time_timestamp - from_time_timestamp)
            )
            * (max_weight - min_weight)
            + min_weight
        )
    )

weighted_operations = operations.withColumn("weight", ...
    interval_weight.cast("float"))
```

4.2 Rule-Based Recommendation

4.2.5 Discrete Timestamp Weights

This is another implementation using the 'timeGenerated' column. As the name suggests, this implementation is a discrete version of the previous method. As seen in listing 4.19, one also has the *num_steps* and *weights* parameters in addition to the common window parameters.

Listing 4.19: Parameters belonging to the Discrete Timestamp Weights method. Code from RuleBased_stepWeights.ipynb.

```
params = {
    'num_steps': num_steps,
    'weights': weights,
    "windowStart": from_time_timestamp,
    "windowEnd": to_time_timestamp,
    "windowSize": interval,
}
```

The *num_steps* parameter is an integer that decides how many buckets the window will be divided into. For example, if the window spans February month and *num_steps* is equal to four, then February will be divided into four buckets, each representing a week. The next parameter, *weights*, is a list representing the weighting for each of these buckets. Since we use four buckets in this example, an input of *weights* = [1, 2, 3, 4] could be used. This will ensure that operations happening in the last week of February will have a weight of 4 compared to operations at the beginning which will have a weight of 1.

Listing 4.20 shows how the timestamps for the different intervals are calculated based on the input parameters. It also shows how the weights are assigned to the operations table.

Listing 4.20: Code to calculate the timestamps for the different intervals inside the window as well. It also shows how the weights are assigned in the Discrete Timestamp Weights method. Code from RuleBased_stepWeights.ipynb.

```
interval_duration = round((to_time_timestamp - ...
    from_time_timestamp)/num_steps)
start = round(from_time_timestamp)

intervals = []
for i in range(num_steps):
    intervals.append((start, start + interval_duration))
    start = start + interval_duration
```

4.2 Rule-Based Recommendation

```
def assign_weight(timestamp):
    for i in range(len(intervals)):
        if timestamp >= intervals[i][0] and timestamp < intervals[i][1]:
            return weights[i]
    return None

assign_weight_udf = F.udf(assign_weight, FloatType())
weighted_operations = operations.withColumn("weight", ...
    assign_weight_udf("timeGenerated"))
```

These weights are then grouped in the same way as the other methods, and results are written to the method tables.

4.2.6 Run Time Weights

The parameters of the Run Time Weights method include the max and min weight and the *max_time* parameter. The *max_time* parameter is how long (seconds) the operation's physical plan's run time has to be in order for the operation to get assigned the *max_weight*.

Listing 4.21: Parameters belonging to the Run Time Weights method. Code from Rule-Based_runTimeWeightedCounts.ipynb.

```
params = {
    "max_weight": max_weight,
    "min_weight": min_weight,
    "max_time": max_time,
    "windowStart": from_time_timestamp,
    "windowEnd": to_time_timestamp,
    "windowSize": interval,
}
```

In the previous method, Timestamp Weights, the *min_weight* and *max_weight* parameters were used at the beginning and end of the window of interest to assign weights based on when the operation took place inside that interval. The weights are now used to weigh the operations based on their physical plan's run time. More specifically: where in the interval from 0 to *max_time* seconds it lands.

In order to get the run time of the operation's physical plans, we need to join the operations table with the queries table, this can be done using the key 'physicalPlanKey'.

4.3 Statistical Calculations

Listing 4.22: Joining the operations and queries table to get the run time information into the operations table. Code from RuleBased_runTimeWeightedCounts.ipynb.

```
run_times = spark.sql("select physicalPlanKey, duration_ms from queries")
operations = operations.join(run_times, on='physicalPlanKey')
```

After joining the tables, the weights are calculated similarly to the timestamp weights: being outside of the interval $[0, max_time]$ will result in the min and max weights. Data points inside the interval will use linear interpolation to calculate the respective weights. Calculations can be seen in listing 4.23.

Listing 4.23: Showing how the run time weights are calculated. Code from RuleBased_runTimeWeightedCounts.ipynb.

```
run_time_weight = F.when(
    F.col("duration_ms") >= max_time_ms, max_weight
).otherwise(
    F.when(F.col("duration_ms") <= 0, min_weight).otherwise(
        (
            F.col("duration_ms") / max_time_ms
        )
        * (max_weight - min_weight)
        + min_weight
    )
)
weighted_operations = operations.withColumn("weight", ...
    run_time_weight.cast("float"))
```

The resulting *weighted_operations* DF is then grouped on each distinct column in 'columnName' where the operation weights are summed. This result is sent to the *method_results* table, and the highest scoring column for each table is sent to the *method_recommendation* table.

4.3 Statistical Calculations

In the previous sections, we saw how data was extracted from logs and used to perform column recommendations. To evaluate the effectiveness of these methods, we wanted to calculate some simple statistics.

4.3 Statistical Calculations

The notebook responsible for the statistics has three different input parameters: *start_time*, *end_time*, and *interval*. The *interval* variable is used to specify how many weeks each statistical period inside the window should be. We join the operations and queries tables so that we can use the run time of the physical plans. Code for handling the interval logic can be seen in 4.24.

Listing 4.24: Shows code used for creating intervals and assigning them to the data. Code from `statisticsCalc.ipynb`.

```
intervals = []
for time in range(int(start_time_unix), int(end_time_unix), ...
    int(interval_unix)):
    intervals.append((time, (time + int(interval_unix))))

def assign_group(timestamp):
    for i in range(len(intervals)):
        if timestamp >= intervals[i][0] and timestamp < intervals[i][1]:
            return float(intervals[i][0])
    return None

assign_group_udf = F.udf(assign_group, DoubleType())
combined_df = combined_df.withColumn("interval_group", ...
    assign_group_udf("time_start"))
```

After the intervals were decided, we performed the statistical calculations. Statistics were created based on run time on both column and table levels. The calculations were carried out using a `groupBy` operator, first on the columns or tables (depending on the statics type) and then on each of the intervals. Next, we performed an aggregation followed by the different statistical functions seen in listing 4.25.

Listing 4.25: Shows code used for aggregate calculations on columns. Code from `statisticsCalc.ipynb`.

```
column_stats = combined_df.
groupBy(combined_df.columnName.alias("colName"),
combined_df.interval_group.alias("int_group")).agg(
    F.mean("duration_ms").alias("avg_duration_ms"),
    F.max("duration_ms").alias("max_duration_ms"),
    F.min("duration_ms").alias("min_duration_ms"),
    F.sum("duration_ms").alias("sum_duration_ms"),
    F.variance("duration_ms").alias("variance_duration_ms"),
    F.stddev("duration_ms").alias("stddev_duration_ms"),
    F.kurtosis("duration_ms").alias("kurtosis_duration_ms")
)
```

4.4 Pipeline

Finally, two new tables called "query_column_stats" and "query_table_stats" were made, storing the statistics for each respective kind.

4.4 Pipeline

The notebooks explained in this section are integrated into Databricks' version of pipelines called Workflows. A visualization of how the notebooks have been connected can be seen in figure 4.4.1.

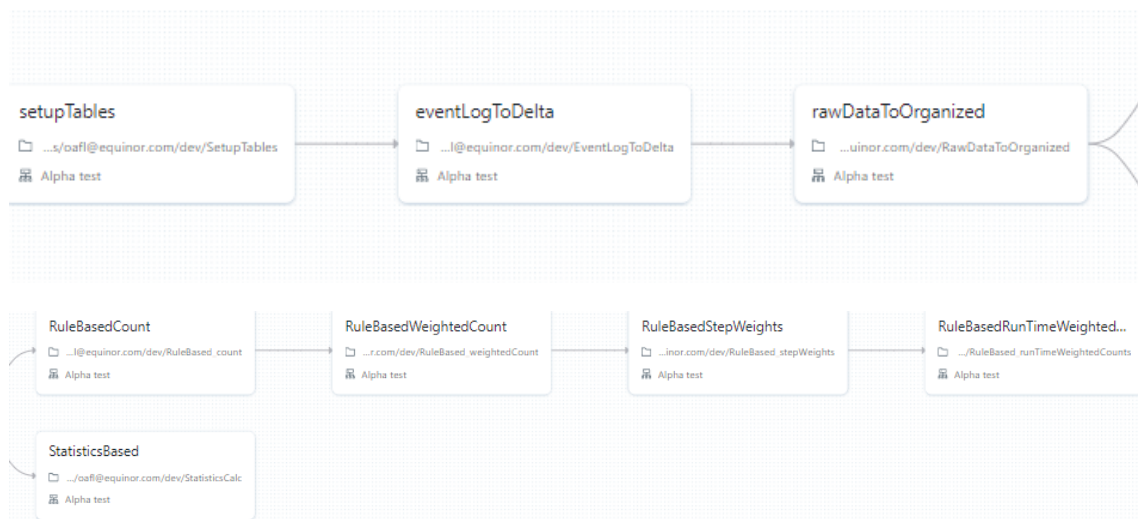


Figure 4.4.1: Figure shows how the pipeline is presented in Databricks Workflows.

Chapter 5

Experiments and Results

This section presents the results gathered from our two experiments. In the first experiment, our focus was to test the capabilities of our recommendation methods on various query patterns and examine the impact of partitioning and Z-ordering on relevant columns. In the second experiment, we simulated a real use case of our recommendation system and compared its run times to an unoptimized simulation without the use of partitions or Z-order. For both experiments, we follow the theory explained in the background section about partitions and Z-ordering, specifically about partitioning on categorical columns while using Z-order for high-cardinality columns.

The experiments were carried out in Databricks with a Spark cluster using Databricks version 12.2 with Apache Spark 3.3.2 and Scala 2.12. The cluster utilized 2-6 workers with 28 GB memory and 8 cores (Standard_DS4_v2). The driver type was Standard_DS4_v2 which translates to 28 GB and 8 cores. The experiments were conducted using the TPC-H Benchmark [33] data⁶. Listing 5.1 shows the method parameters used in both of the experiments⁷.

⁶Size (MB) of the tables were: lineitem \approx 1097, orders \approx 286, customer \approx 61, part \approx 31, supplier \approx 4, nation \approx 0. The choice of data size is discussed in the "Discussion" section.

⁷The start and end-time parameters are set to the timestamps when the first and last query in the group were run.

5.1 Method Experiment

Listing 5.1: Shows the methods' input parameters used for the experiments. An detailed explanation of what the parameters does is given in section 2.10.

```
TimestampMinWeight = "0"  
TimestampMaxWeight = "2"  
  
DiscreteTimestampNumSteps = "4"  
DiscreteTimestampWeights = "0.25, 0.5, 0.75, 1"  
  
RunTimeMinWeight = "0"  
RunTimeMaxWeight = "2"  
RunTimeMaxTime = "4"
```

5.1 Method Experiment

For this experiment, we utilized two TPC-H tables: lineitem and orders. The primary objective of this experiment was to evaluate the capabilities of our methods in detecting different query patterns. We also investigated the impact of Z-ordering versus partitioning on the performance of these query patterns. Although the data used in this experiment is of small size, we conducted each query group five times and calculated the average run time to ensure consistency. It is important to note that this pilot experiment serves as a preliminary investigation before a more realistic simulation in the subsequent experiment. For those interested in reproducing our experiments, the queries used in this experiment can be found in the Supplementary Information in Section A.4.

5.1.1 Query Group 1

Query group 1 evaluates the performance of the recommendation methods under sudden shifts in query patterns. The queries in this group operate on the lineitem table, with 10 queries filtering on the 'l_orderkey' column followed by 12 queries filtering on the 'l_shipmode' column. Table 5.1 displays the top column recommendations from each method.

5.1 Method Experiment

Method Name	Table	Recommended Column
Simple Count	lineitem	l_shipmode
Timestamp Weights	lineitem	l_shipmode
Discrete Timestamp Weights	lineitem	l_shipmode
Run Time Weights	lineitem	l_shipmode

Table 5.1: Column recommendations for group 1.

Average Group Run Time (s)	lineitem_table
6.287	Partition: 'l_shipmode'
8.650	Z-order: 'l_orderkey'
13.712	No optimization

Table 5.2: Average group 1 run times for different optimizations.

Table 5.2 shows the run times of all queries in group 1 with three different optimizations. The unoptimized run had an average run time of 13.712 seconds. Partitioning on 'l_shipmode' achieved an average run time of 6.287 seconds, resulting in a significant improvement of 7.425 seconds (54%) compared to the unoptimized run. Z-ordering on the 'l_orderkey' column led to an average run time of 8.650 seconds, an improvement of 5.062 seconds (37%) compared to the unoptimized run.

5.1.2 Query Group 2

Query group 2 was designed to evaluate the effectiveness of the Timestamp Weights and Discrete Timestamp Weights methods in considering the recency of queries when recommending columns for optimization. This group only uses the lineitem table and consists of 12 queries that filter on the 'l_shipmode' column, followed by 10 queries that utilize the 'l_orderkey' column. Table 5.3 presents the top recommendations from each method.

5.1 Method Experiment

Method Name	Table	Recommended Column
Simple Count	lineitem	l_shipmode
Timestamp Weights	lineitem	l_orderkey
Discrete Timestamp Weights	lineitem	l_orderkey
Run Time Weights	lineitem	l_shipmode

Table 5.3: Column recommendations for group 2.

Average Group Run Time (s)	lineitem_table
6.646	Partition: 'l_shipmode'
9.200	Z-Order: 'l_orderkey'
12.541	No optimization

Table 5.4: Average group 2 run times for different optimizations.

The run time results for query group 2 are displayed in Table 5.4. Running the group without any optimization techniques resulted in an average run time of 12.541 seconds. Applying Z-ordering on the 'l_orderkey' column reduced the average run time to 9.200 seconds, resulting in a decrease of 3.341 seconds (27%) compared to the unoptimized run. Similarly, partitioning on the 'l_shipmode' column led to an average run time of 6.646 seconds, representing a reduction of 5.895 seconds (47%) compared to the unoptimized approach.

5.1.3 Query Group 3

Query group 3 was designed to evaluate the effectiveness of the Run Time Weights method in prioritizing queries with longer run times. In this group, both the lineitem and the orders table are used. This group consists of 10 straightforward queries that filter on lineitem's 'l_shipmode' column, followed by 3 more complex queries that perform a join between the lineitem and orders table using the 'l_orderkey' and 'o_orderkey' columns. The column recommendations from each method are presented in Table 5.5.

5.2 Recommendation System Experiment

Method Name	Table	Recommended Column
Simple Count	lineitem	l_shipmode
Timestamp Weights	lineitem	l_shipmode
Discrete Timestamp Weights	lineitem	l_shipmode
Run Time Weights	lineitem	l_orderkey
Simple Count	orders	o_orderkey
Timestamp Weights	orders	o_orderkey
Discrete Timestamp Weights	orders	o_orderkey
Run Time Weights	orders	o_orderkey

Table 5.5: Column recommendations for group 3.

Average Group Run Time (s)	lineitem_table	order_table
9.605	Partition: 'l_shipmode'	Z-order: 'o_orderkey'
10.766	Z-order: 'l_orderkey'	Z-order: 'o_orderkey'
13.137	No optimization	No optimization

Table 5.6: Average group 3 run times for different optimizations.

Since 'o_orderkey' is the only column used from the orders table, we will exclusively apply Z-ordering to the order table, except for the unoptimized run where no optimization is applied. Table 5.6 displays the average run times for group 3. The run without optimizations, results in an average run time of 13.137 seconds. By optimizing with Z-ordering on lineitem's 'l_orderkey' column, we achieve an average run time of 10.766 seconds, representing a 2.317 seconds (18%) improvement. Additionally, partitioning on the 'l_shipmode' column yields an average run time of 9.605 seconds, resulting in a reduction of 3.532 seconds (27%).

5.2 Recommendation System Experiment

In this experiment, we want to simulate a more realistic use case of the recommendation system. To simulate realistic queries and to compare results, we took inspiration from

5.2 Recommendation System Experiment

Gou et al. mentioned in section 2.9.8, and chose 8 of the base template queries from the TPC-H: *q1, q4, q6, q10, q14, q15, q18* and *q19* as a base. We used the base queries to build 26 exploratory queries, which are quite similar to the base, and 16 intermediate queries representing more straightforward queries meant to simulate an initial data exploration phase. The intermediate, exploratory, and base queries (in this order) are the collection of 50 queries that are used in this experiment. We define every 10 queries as a batch of queries that are run before updating the partitioning and Z-ordering of the tables.

As our recommendation system is not built to read logs in real-time but instead uses a batch-wise approach for log reading, we needed to do an initial run of the queries to see which columns were recommended after each batch of queries. In this experiment, we employed our four methods, each recommending a column after each batch. In cases where the recommendations differed among the methods, we selected the column that was recommended the most frequently out of the four. Which columns were recommended the most frequently after each batch can be seen in table 5.7.

We chose to perform Z-ordering on all of the recommended columns with the exception of 'n_nationkey', this decision is addressed in the discussion chapter 6.2.2.

Columns	Recomennded Column After Batch 1	Recomennded Column After Batch 2	Recomennded Column After Batch 3	Recomennded Column After Batch 4
orders	o_orderdate	o_custkey	o_orderkey	o_orderkey
customer	c_custkey	c_custkey	c_custkey	c_custkey
lineitem	l_shipdate	l_shipdate	l_orderkey	l_orderkey
supplier				s_suppkey
nation			n_nationkey	n_nationkey
part		p_size	p_size	p_size

Table 5.7: Table shows the most frequently recommended column for each table after each batch of 10 queries.

We ran through all of the queries five times and took the median run time for each query. We did this once using the original tables, and once using the tables optimized by the learned columns from the initial run of the queries. Results for the baseline and the optimized tables can be found in figures 5.2.1, 5.2.2 and 5.2.3. The total run times were 103.23 seconds and 85.59 seconds for the baseline and optimized approach respectively. This is a difference of 17.64s showing a 17.09% reduction in run time.

Because of the short run times and few runs in the experiment, we also calculated a

5.2 Recommendation System Experiment

confidence interval for the differences between the five baseline and optimized runs. We ended up with a 95% confidence interval between [16.75s, 24.67s] seconds reduction and [17.0%, 19.3%] decrease in run time which seems to solidify the 17.64 seconds reduction found when taking the median of the individual queries.

Total Run Time (50 queries) - Baseline vs Optimized

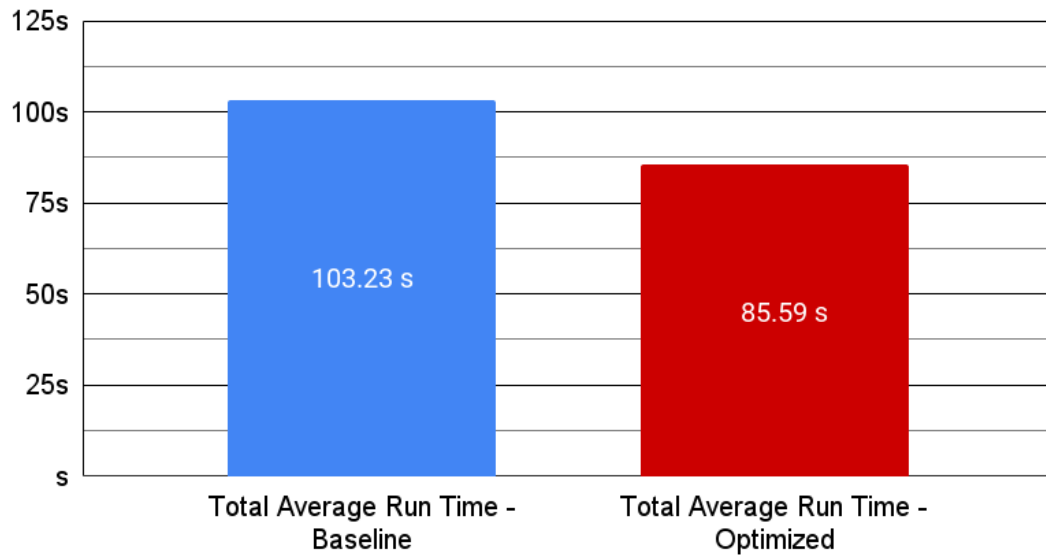


Figure 5.2.1: Comparing the average run times of the baseline and the optimized performance. The baseline is the run with no table optimizations. Optimized is the run after having performed Z-ordering on the recommended columns.

5.2 Recommendation System Experiment

Baseline Query Run Times

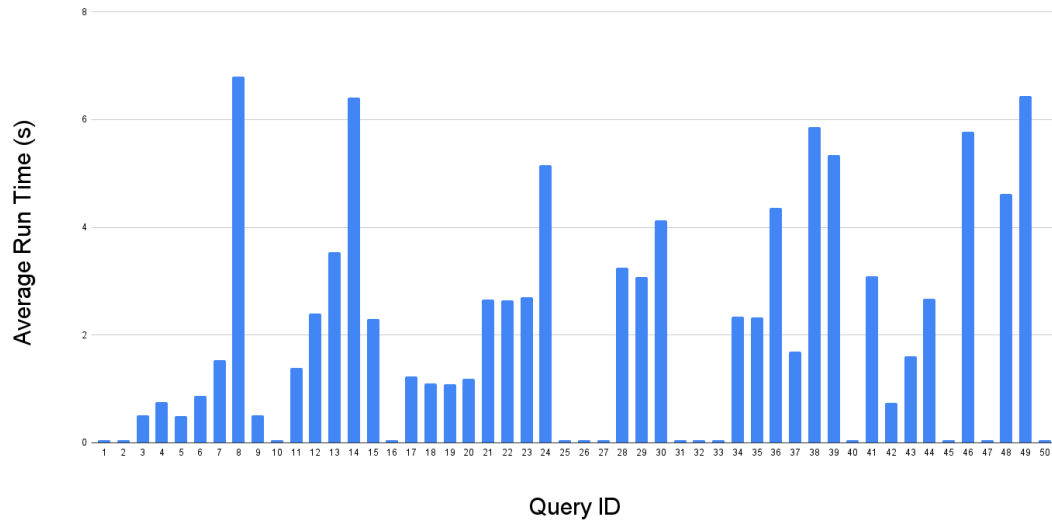


Figure 5.2.2: Median of each query from the baseline runs.

Optimized Query Run Times

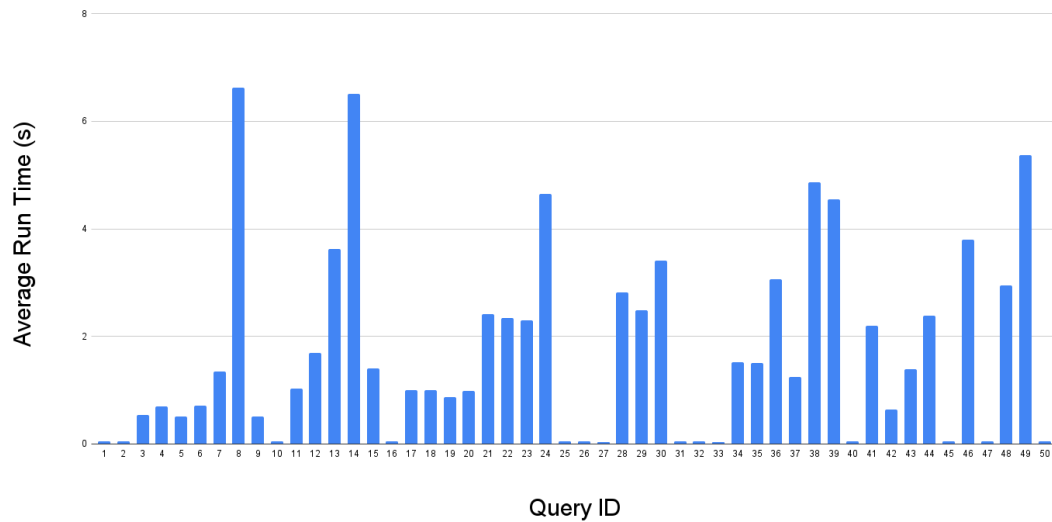


Figure 5.2.3: Median of each query from the optimized runs.

Chapter 6

Discussion

In this section, we will discuss the experiment results, some of the decisions taken during the experiments, and the limitations of our implementations.

6.1 Experiment Validation

Acknowledging that our recommendation system's primary design is to be run and used for column recommendations in a routine fashion, we had to make some adjustments to prepare it for experiments needing frequent reading of the event logs. We had to adjust it for experiments. Due to time constraints and practical reasons, we opted for smaller data sets. This decision was also influenced by the additional time required for Spark to store event logs in the designated location.

As mentioned in the method experiment; this is a preliminary experiment with the main purpose being to see if the methods are able to detect the query patterns in groups 1, 2, and 3. We also wanted to see if there were any clear differences between the performance of optimization techniques; partitioning and Z-ordering. To look for the difference in performance between the two optimization techniques, we ran each of the query groups five times and used the runs' mean value as our recorded run times.

In the final experiment, we maintained consistency by running the complete set of 50 queries five times for both the baseline and optimized approaches. For each query, we

6.2 Discussion of Results

calculated the median duration from the five runs and summed them up. To verify the calculated difference of 17.64 seconds, we computed a confidence interval using a [student-t distribution](#). This approach was chosen due to the relatively small number of data points available for analysis. The resulting 95% confidence interval provides a realistic range ([16.75 seconds, 24.67 seconds]) of values, matching our calculated difference between the baseline and optimized approach.

6.2 Discussion of Results

In this section, we analyze the results of our experiments and discuss the findings. The recommendation methods employed in our experiments look for different query characteristics and weigh the operations according to the input parameters shown in listing 5.1.

In the following subsection, we discuss the methods' strengths and weaknesses highlighted by the different query groups from the method experiment, after that, we analyze how the run times for different query groups are effected by the different optimization techniques: partitioning and Z-ordering. Whereas in the last subsection, we look at the results from the last experiment, discussing the performance of the recommendation system as a whole.

6.2.1 Method Experiment

6.2.1.1 Recommendations

Group 1 contains 10 queries filtering on lineitem's 'l_orderkey' column followed by 12 queries using the 'l_shipmode' column. We see that all methods recommend shipmode as the most important column. This is expected, as the most used column is also the most recent one, giving it a larger weight for both the Simple Count method and the methods taking time-of-execution into account. As long as the queries utilizing the orderkey column do not have a longer run time, it makes sense that none of the methods chose this as the most important column.

Group 2 uses the same queries as in the first group but in the opposite order. 12 queries using the shipmode column followed by 10 queries filtering on the 'l_orderkey' column. This experiment was made to see if the methods evaluating queries using time-of-execution could prioritize the orderkey column even though it was outnumbered 10 to 12. As seen

6.2 Discussion of Results

in table 5.3, both the Timestamp Weight and its discrete implementation, Discrete Timestamp Weights recommended the 'l_orderkey' column. In ETL platforms where query trends often evolve over time, these methods' predictive nature can be used effectively by appropriately tuning the interval length and min/max weights (explained in section 2.10.0.2) to match the query trend volatility of the platform.

Group 3 has 10 queries filtering on lineitem's 'l_shipmode' column and three queries that utilize join operations on the 'orderkey' column between the lineitem and the orders table. The 'o_orderkey' column is the only column used from the orders table and is therefore recommended by all four methods for this table.

Even though there are three queries filtering on the 'l_orderkey' column and 10 queries filtering on the 'l_shipmode' column in Group 3, the Run Time Weights method recommends the 'l_orderkey' column. This preference is due to the method assigning a larger weight to queries with longer run times, which includes the 3 queries utilizing the 'l_orderkey' column. This is because the Run Time Weights method favors queries with longer run times.

When using the Run Time Weights method to perform recommendations on an ETL platform, using a low value⁸ for the *max_time* parameter will give larger weights to queries with a shorter run time. However, it is not possible to differentiate weights for queries with run times above the *max_time* threshold. When determining an appropriate value for the *max_time* parameter, we suggest analyzing the query run times of the specific platform.

6.2.1.2 Query Group Run Times

Run times in both query group 1 and 2 show that partitioning on the shipmode column reduces the run time more than Z-order does when used on the orderkey column. The partitioning gives a speed increase of 54% and 47% compared to the 37% and 27% when using the Z-order for group 1 and 2 respectively. Run times for group 3, displayed in table 5.6, show that the run time for partitioning on 'l_shipmode' and Z-order on 'l_orderkey' are very similar (9.6s and 10.8s) compared to the other groups. This could indicate that Z-ordering increases the efficiency of the three joins almost as much as the partitions on shipmode optimize the 10 queries using the shipmode column. Based on the tests, it looks like both partitioning and Z-ordering are good options for optimizing queries. Which one

⁸This is highly subjective as different platforms have different interpretations of what is a short or long run time

6.2 Discussion of Results

is superior seems to depend on the characteristics of the table and column.

6.2.2 Recommendation System Experiment

After the initial run of the 50 queries, we see that all of the recommended columns after each batch are 'key' or 'date' columns except for the 'p_size' column from the part table. We ended up performing Z-ordering on every⁹ recommended column because no categorical low cardinality columns were chosen. Low cardinality date columns can be a good option used for partitioning, either on yyyy-mm-dd format or split into a three-level-partition with columns: year, month, and date. If the tables had been larger, partitioning on dates would have been a potential optimization to explore. However, based on our experience with the small tables used in the experiment, we chose Z-ordering as the better alternative for the date columns. Also, when discussing results from query group 3 in the previous experiment, we found that using Z-ordering on query group 3 gave a more similar optimization percentage compared to the partition counterpart, which may imply Z-ordering is preferred when working with joins on id columns.

When performing the experiment, we used five runs and based the query run times on these runs. We planned to take the mean of the values. However, we noticed some outliers with longer run times for the first run for both the baseline and the optimized run, which is why we decided to take the median instead of the mean as we did in the first experiment. Studying the output from tables 5.2.2 and 5.2.3 shows a similar run time for the first 10 queries where none of the runs had any optimization, followed by a reduction in run time for the optimized run in the subsequent queries.

Comparing the experiment's results with the work of Guo et al., who performed experiments on a larger scale with a different partition system but on the same tables, reveals some interesting insights. Even though the data scale differs significantly, we observe a similar trend between the baseline and optimized runs. In our experiment, we managed to get an improvement of 17% compared to runs without partitions, whereas Guo et al. achieved a 19% and 20% improvement for their experiments. While our study's results may not directly align due to variations in data, by incorporating their work, we can better understand the effectiveness and performance gains achievable by systems optimizing data retrieval in Spark.

In contrast to Guo et al., our system has been tested on an ETL platform, namely

⁹We decided not to do any optimization on the nation table since it only contained 25 rows of data.

6.3 Limitations in Methodology

Databricks, and has easy integration with ETL platforms using Spark as it only requires access to the Spark event logs.

6.3 Limitations in Methodology

The solution proposed in this thesis extracts insightful information from Spark event logs to determine important columns for partitioning and Z-ordering on tables. Still, it has its limitations. When parsing a Spark physical plan, we needed to bind columns used in operations such as 'Filter' to their corresponding table. The solution was to use the 'scan parquet' part of the physical plan. The 'scan parquet' operation represents Spark's plan of how to read from parquet files and contains table names and corresponding internal Spark ids for columns. Using Spark's internal ids enabled us to create a dictionary, which could later be used to bind columns to their respective tables. Although this worked for connecting columns to tables, it limited our solution only to work on tables stored in parquet format.

Chapter 7

Future Work

In this chapter, we present future work that can be used to build upon and expand on our findings.

7.1 Enhanced Parsing of the Physical Plan

Presented in the implementation section and seen in code snippet 4.11, expanding the parsing of Spark's physical plans is quite feasible. Adding new regex settings can improve the current solution and make it compatible with tables stored in file formats other than the parquet format. Additionally one could parse information on what columns are used to join tables and develop a method able to utilize this information.

7.2 Method with Multiple Weights

The current recommendation methods use count, time, and run time as metrics, but separately. A new approach could combine these metrics into a new method to improve the accuracy of finding the best recommendations. It could also use new features extracted from the physical plan, such as information about joins, mentioned in the previous section.

7.3 Automation

7.3 Automation

The current implementation presents the user with a recommended column from each of the four methods. Should the methods not be unanimous, the user would have to make an educated guess on which column is the preferred choice. Implementing a process to make a final decision based on the output from the four methods will eliminate the need for an educated guess. This process can be implemented into a notebook with some additional benefits.

To decide if partitioning or Z-ordering of a column should occur, we would calculate a fraction for the number of distinct values in the column and decide on Z-ordering or partitioning based on this value. Another improvement could be to check the size of the tables to see if it is necessary to have partitioning or Z-ordering at all.

Further, it is possible to have partitioning and Z-ordering on a single table or have partitions on multiple columns simultaneously. Instead of recommending only one column, our solution could be changed to recommend multiple columns and automatically partition or Z-order these columns depending on the criterion discussed in the previous paragraph.

7.4 Multiple Platforms

As mentioned in the Challenges section (1.5), the plan was to extend our practical work of log analyzing and our parsing solution into Azure Synapse Analytics, but we ran out of time. Instead, we propose the extension into Azure Synapse Analytics as future work to be explored.

7.5 Dashboard

As mentioned in the challenges section; our initial plan was to build a dashboard giving an overview of column recommendations for each table, as well as the run time statistics calculated. This could be made to give a better overview of which columns are recommended and see if one is able to pick up any trends in run time.

Chapter 8

Conclusion

In this thesis, we proposed a column recommendation system based on Spark SQL which highlights important columns to use for partitioning and Z-ordering. Our system employs four methods to recommend columns to use for optimizing data retrieval in Spark. Having Spark event logs as the only data input makes our recommendation system easy to integrate and a viable option for multiple ETL platforms. The results from the experiments show that using the system's output will optimize queries run on the platform. We will end by taking a look at the research questions introduced at the beginning of the thesis.

1. **Can Spark's physical plans be used to optimize partitioning and Z-ordering on tables?**

Yes, the physical plans are Spark's way of describing how queries are executed on a low level in detail. The information in the physical plan gives a deeper look into what type of operations Spark uses, which results in an advantage in the optimization process.

2. **Can rule-based column recommendation methods be used to improve the partitioning and Z-ordering of tables?**

Yes, from the results of the first experiment, we can see that the methods are able to detect the query patterns and query characteristics presented. In the first experiment, we see that the usage of partitioning and Z-ordering gives a significant reduction in run times.

3. **Can the use of continuous updates of partitioning and Z-order be used to optimize queries in an ETL platform?**

Conclusion

Yes, in our last experiment, using the column recommendation system provided a 17% optimization in run time compared to the simulation without *Z*-order or partitions.

Bibliography

- [1] Microsoft Azure. What is data lake? <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-data-lake/>. Accessed: 06/13/23.
- [2] Micheal Berk. Demystifying the parquet file format. <https://towardsdatascience.com/demystifying-the-parquet-file-format-13adb0206705>, 08 2022. Accessed: 04/13/23.
- [3] Luc Bouganim. *Data Skew*, pages 634–635. Springer US, Boston, MA, 2009. Editor: LIU, LING and ÖZSU, M. TAMER.
- [4] Boudewijn Braams. The parquet format and performance optimization opportunities boudewijn braams (databricks). https://www.youtube.com/watch?v=1j8SdS7s_NY&ab_channel=Databricks, 10 2019. Accessed: 04/13/23.
- [5] J. Damji, B. Wenig, D. Lee, T. Das, and an O’Reilly Media Company Safari. *Learning Spark, 2nd Edition*. O’Reilly Media, Incorporated, 2 edition, 2020.
- [6] Databricks. Apache spark. <https://www.databricks.com/glossary/what-is-apache-spark>. Accessed: 06/13/2023.
- [7] Databricks. Data skipping with z-order indexes for delta lake. <https://docs.databricks.com/delta/data-skipping.html>. Accessed: 06/13/2023.
- [8] Databricks. Datasets. <https://www.databricks.com/glossary/what-are-datasets>. Accessed: 06/13/2023.
- [9] Databricks. Mapreduce. <https://www.databricks.com/glossary/mapreduce>. Accessed: 06/13/2023.
- [10] Databricks. Spark api. <https://www.databricks.com/glossary/spark-api>. Accessed: 06/13/2023.

BIBLIOGRAPHY

- [11] Databricks. Transformations. <https://www.databricks.com/glossary/what-are-transformations>. Accessed: 06/13/2023.
- [12] Databricks. Tungsten. <https://www.databricks.com/glossary/tungsten>. Accessed: 06/13/2023.
- [13] Databricks. Understanding and improving code generation. <https://www.youtube.com/watch?v=wVs1FZyKXMY>, 2020. Accessed: 06/13/23.
- [14] Databricks. What is databricks workflows? <https://docs.databricks.com/workflows/index.html>, 2023. Accessed: 06/13/23.
- [15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [16] Chenghao Guo, Zhigang Wu, Zhenying He, and X Sean Wang. An adaptive data partitioning scheme for accelerating exploratory spark sql queries. In *Database Systems for Advanced Applications: 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part I 22*, pages 114–128. Springer, 2017.
- [17] Bennie Haelen and Dan Davis. *Delta Lake: Up and Running*. O’Reilly Media, Inc., 2023. Ebook O’Reilly edition.
- [18] Zichun Huang, Wenguo Wei, and Guiyuan Xie. Load balancing mechanism based on linear regression partition prediction in spark. *Journal of Physics: Conference Series*, 1575(1):012109, jun 2020.
- [19] Sital Kedia and Gaoxiang Liu. Tuning apache spark for large scale workloads - sital kedia & gaoxiang liu. <https://youtu.be/5dgaOUT4RI8>, 06 2017. Accessed: 06/12/2023.
- [20] Romeo Kienzler. *Mastering Apache Spark 2.x - Second Edition*. Packt Publishing, 2017. Ebook O’Reilly edition.
- [21] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd Edition*. Wiley, 3 edition, 2013. Ebook O’Reilly edition.
- [22] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. Rios: Runtime integrated optimizer for spark. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 275–287, 2018.
- [23] Apache Parquet. File format. <https://parquet.apache.org/docs/file-format/>, 2022. Accessed: 06/13/23.

BIBLIOGRAPHY

- [24] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M. Mustafa Rafique, and Ali R. Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 110–119, 2016.
- [25] David Buchaca Prats, Felipe Albuquerque Portella, Carlos H.A. Costa, and Josep Lluís Berral. You only run once: Spark auto-tuning from a single run. *IEEE Transactions on Network and Service Management*, 17:2039–2051, 12 2020.
- [26] Python. Pathlib — object-oriented filesystem paths. <https://docs.python.org/3/library/pathlib.html>. Accessed: 06/13/23.
- [27] Surendran Rajendran, Osamah Ibrahim Khalaf, Youseef Alotaibi, and Saleh Alghamdi. Mapreduce-based big data classification model using feature subset selection and hyperparameter tuned deep belief network. *Scientific Reports*, 11(1):24138, 2021.
- [28] Nick Samoylov. *Introduction to Programming*. Packt Publishing, 2018. Ebook O’Reilly edition.
- [29] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. Autoexecutor: Predictive parallelism for spark sql queries. *Proceedings of the VLDB Endowment*, 14:2855–2858, 2021.
- [30] Apache Spark. Mllib is apache spark’s scalable machine learning library. <https://spark.apache.org/mllib/>. Accessed: 06/13/2023.
- [31] Apache Spark. Physical Planning Catalyst. <https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala>. Accessed: 04/14/23.
- [32] Apache Spark. Spark sql, dataframes and datasets guide. <https://spark.apache.org/docs/3.3.2/sql-programming-guide.html>. Accessed: 06/13/2023.
- [33] TPC. Tpc-h vesion 2 and version 3. <https://www.tpc.org/tpch>. Accessed: 06/13/2023.
- [34] David Vrba. Mastering query plans in spark 3.0. <https://towardsdatascience.com/mastering-query-plans-in-spark-3-0-f4c334663aa4>, 07 2020. Accessed: 04/12/23.
- [35] Jinhan Xin, Kai Hwang, and Zhibin Yu. Locat: Low-overhead online configuration auto-tuning of spark sql applications. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 674–684, 6 2022.

BIBLIOGRAPHY

- [36] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

Appendix A

Supplementary Information

A.1 Github repository and Dataset

Our code is available at our GitHub repository:
<https://github.com/Olaaflo/SparkOptimization>

A.2 Code

Listing A.1: Schema used to read the raw event logs. Types are imported from pyspark.sql.types. Code from rawDataToOrginaized.ipynb.

```
schema = ArrayType(  
    StructType(  
        [  
            StructField("Event", StringType(), True),  
            StructField("SparkContext Id", StringType(), True),  
            StructField("Stage Info", StringType(), True),  
            StructField("Task Info", StringType(), True),  
            StructField("Stage ID", StringType(), True),  
            StructField("Task End Reason", StringType(), True),  
            StructField("Stage IDs", StringType(), True),  
            StructField("Stage Attempt ID", StringType(), True),  
            StructField("Completion Time", StringType(), True),  
            StructField("time", StringType(), True),
```


A.3 Figures and Illustrations

```
StructField("errorMessage", StringType(), True),
StructField("Task Executor Metrics", StringType(), True),
StructField("Timestamp", StringType(), True),
StructField("executionId", StringType(), True),
StructField("Job Result", StringType(), True),
StructField("Stage Infos", StringType(), True),
StructField("details", StringType(), True),
StructField("Task Metrics", StringType(), True),
StructField("physicalPlanDescription", StringType(), True),
StructField("modifiedConfigs", StringType(), True),
StructField("Submission Time", StringType(), True),
StructField("rootExecutionId", StringType(), True),
StructField("Spark Version", StringType(), True),
StructField("Rollover Number", StringType(), True),
StructField("sparkPlanInfo", StringType(), True),
StructField("Job ID", StringType(), True),
StructField("Task Type", StringType(), True),
StructField("description", StringType(), True),
StructField("Properties", StringType(), True),
StructField("accumUpdates", StringType(), True),
    ]
)
)
```

A.3 Figures and Illustrations

Figure A.3.1 illustrates an entity relationship diagram for the tables used in this thesis.

A.3 Figures and Illustrations



Figure A.3.1: Entity relationship model of tables.

A.3 Figures and Illustrations

columnName	databaseName	executionId	operationName	physicalPlanKey	eventlogKey	tableName	timeGenerated	operationId
sparkContextID	default	568	PushedFilters	-2113573252	848484182	eventlog_raw	1681726704046	68719476746
lastModified	default	568	PushedFilters	-2113573252	848484182	eventlog_raw	1681726704046	68719476747
clusterInstanceID	default	568	PushedFilters	-2113573252	848484182	eventlog_raw	1681726704046	68719476748
sparkContextID	default	568	PushedFilters	-2113573252	848484182	eventlog_raw	1681726704046	68719476749
lastModified	default	568	PushedFilters	-2113573252	848484182	eventlog_raw	1681726704046	68719476750
clusterInstanceID	default	568	PushedFilters	-2113573252	848484182	eventlog_raw	1681726704046	68719476751
eventlogKey	default	397	PushedFilters	-2012970020	1207902865	eventlog_raw	1681740276449	68719476771
sparkContextID	default	1512	PushedFilters	-2010522258	-1415253976	eventlog_raw	1683114053722	68719476777
lastModified	default	1512	PushedFilters	-2010522258	-1415253976	eventlog_raw	1683114053722	68719476778
clusterInstanceID	default	1512	PushedFilters	-2010522258	-1415253976	eventlog_raw	1683114053722	68719476779
sparkContextID	default	1512	PushedFilters	-2010522258	-1415253976	eventlog_raw	1683114053722	68719476780
lastModified	default	1512	PushedFilters	-2010522258	-1415253976	eventlog_raw	1683114053722	68719476781
clusterInstanceID	default	1512	PushedFilters	-2010522258	-1415253976	eventlog_raw	1683114053722	68719476782
timeGenerated	default	1240	PushedFilters	-1971355054	459219628	operations	1683111507298	68719476798
operationName	default	1240	PushedFilters	-1971355054	459219628	operations	1683111507298	68719476799
sparkContextID	default	238	PushedFilters	-1970545870	679725161	eventlog_raw	1681336365352	68719476805
lastModified	default	238	PushedFilters	-1970545870	679725161	eventlog_raw	1681336365352	68719476806
clusterInstanceID	default	238	PushedFilters	-1970545870	679725161	eventlog_raw	1681336365352	68719476807

Figure A.3.2: Preview of the operations table.

Figure A.3.3 is a preview of the method_results table, in this case, generated from the Simple Count method.

A.3 Figures and Illustrations

databaseName	tableName	columnName	methodValue	isPartitioned	runId
default	eventlog_raw	clusterInstanceID	534.0	false	5
default	eventlog_raw	eventlogKey	44.0	false	5
default	eventlog_raw	lastModified	535.0	false	5
default	eventlog_raw	sparkContextID	534.0	false	5
default	method_recommenda...	tableName	3.0	false	5
default	method_results	tableName	3.0	false	5
default	operations	columnName	3.0	false	5
default	operations	databaseName	4.0	false	5
default	operations	eventlogKey	61.0	false	5
default	operations	operationId	10.0	false	5
default	operations	operationName	323.0	false	5
default	operations	physicalPlanKey	10.0	false	5
default	operations	tableName	5.0	false	5
default	operations	timeGenerated	344.0	false	5
default	physical_plan_keys	physicalPlanKey	90.0	false	5
default	queries	physicalPlanKey	76.0	false	5
default	queries	time_end	19.0	false	5
default	queries	time_start	20.0	false	5

Figure A.3.3: Preview of the method_results table built from the Simple Count method.

databaseName	tableName	columnName	methodValue	isPartitioned	runId
default	eventlog_raw	lastModified	535.0	false	5
default	method_recommendations	tableName	3.0	false	5
default	method_results	tableName	3.0	false	5
default	operations	timeGenerated	344.0	false	5
default	physical_plan_keys	physicalPlanKey	90.0	false	5
default	queries	physicalPlanKey	76.0	false	5

Figure A.3.4: Preview of the method_recommendations table created from the method_results table (figure A.3.3).

A.4 Tables

A.4 Tables

Avg Run Time (s)	Group ID	Lineitem Table Info	Orders Table Info
13.712	1	Partitions: null, Z-Order: null	Partitions: null, Z-Order: null
13.137	3	Partitions: null, Z-Order: null	Partitions: null, Z-Order: null
12.541	2	Partitions: null, Z-Order: null	Partitions: null, Z-Order: null
10.766	3	Partitions: null, Z-Order: l_orderkey	Partitions: null, Z-Order: o_orderkey
9.605	3	Partitions: l_shipmode, Z-Order: null	Partitions: null, Z-Order: o_orderkey
9.200	2	Partitions: null, Z-Order: l_orderkey	Partitions: null, Z-Order: o_orderkey
8.650	1	Partitions: null, Z-Order: l_orderkey	Partitions: null, Z-Order: o_orderkey
6.646	2	Partitions: l_shipmode, Z-Order: null	Partitions: null, Z-Order: o_orderkey
6.287	1	Partitions: l_shipmode, Z-Order: null	Partitions: null, Z-Order: o_orderkey

Table A.1: Run times to complete the queries for each group after having performed optimizations on the lineitem and orders tables.

A.4 Tables

ID	Name	Table	Col Rec	Internal Method Val	Group ID
3	Discrete Timestamp Weights	lineitem	l_shipmode	9.5	1
4	Run Time Weights	lineitem	l_shipmode	3.3375	1
2	Timestamp Weights	lineitem	l_shipmode	16.179607	1
1	Simple Count	lineitem	l_shipmode	11	1
7	Discrete Timestamp Weights	lineitem	l_orderkey	8	2
8	Run Time Weights	lineitem	l_shipmode	3.3975	2
6	Timestamp Weights	lineitem	l_orderkey	13.733679	2
5	Simple Count	lineitem	l_shipmode	12	2
11	Discrete Timestamp Weights	orders	o_orderkey	2.25	3
11	Discrete Timestamp Weights	lineitem	l_shipmode	3.5	3
12	Run Time Weights	orders	o_orderkey	4.1675	3
12	Run Time Weights	lineitem	l_orderkey	4.1675	3
10	Timestamp Weights	orders	o_orderkey	3.7059548	3
10	Timestamp Weights	lineitem	l_shipmode	4.468995	3
9	Simple Count	orders	o_orderkey	3	3
9	Simple Count	lineitem	l_shipmode	10	3

Table A.2: This table shows the recommended columns given from each of the methods for each of the query groups.

A.4 Tables

Group ID	Query ID	Query Text
1	1	SELECT * FROM lineitem WHERE l_orderkey = 1000;
1	2	SELECT * FROM lineitem WHERE l_orderkey > 5000;
1	3	SELECT * FROM lineitem WHERE l_orderkey BETWEEN 200 AND 400;
1	4	SELECT * FROM lineitem WHERE l_orderkey <> 50;
1	5	SELECT * FROM lineitem WHERE l_orderkey IN (10, 20, 30);
1	6	SELECT * FROM lineitem WHERE RIGHT(l_orderkey, 1) = '5';
1	7	SELECT * FROM lineitem WHERE l_orderkey LIKE '%3%';
1	8	SELECT * FROM lineitem WHERE l_orderkey > 10000 AND l_orderkey < 2000;
1	9	SELECT * FROM lineitem WHERE l_orderkey LIKE '1%';
1	10	SELECT * FROM lineitem WHERE l_orderkey IS NULL;
1	11	SELECT * FROM lineitem WHERE l_shipmode = 'AIR';
1	12	SELECT * FROM lineitem WHERE l_shipmode = 'FOB';
1	13	SELECT * FROM lineitem WHERE l_shipmode = 'MAIL';
1	14	SELECT * FROM lineitem WHERE l_shipmode = 'RAIL';
1	15	SELECT * FROM lineitem WHERE l_shipmode = 'SHIP';
1	16	SELECT * FROM lineitem WHERE l_shipmode = 'TRUCK';
1	17	SELECT * FROM lineitem WHERE l_shipmode = 'AIR' OR l_shipmode = 'FOB';
1	18	SELECT * FROM lineitem WHERE l_shipmode = 'FOB' OR l_shipmode = 'MAIL';
1	19	SELECT * FROM lineitem WHERE l_shipmode = 'MAIL' OR l_shipmode = 'RAIL';
1	20	SELECT * FROM lineitem WHERE l_shipmode = 'RAIL' OR l_shipmode = 'SHIP';
1	21	SELECT * FROM lineitem WHERE l_shipmode = 'SHIP' OR l_shipmode = 'TRUCK';
1	22	SELECT * FROM lineitem WHERE l_shipmode = 'TRUCK' OR l_shipmode = 'AIR';

Table A.3: Queries in group 1.

A.4 Tables

Group ID	Query ID	Query Text
2	1	SELECT * FROM lineitem WHERE l_shipmode = 'AIR';
2	2	SELECT * FROM lineitem WHERE l_shipmode = 'FOB';
2	3	SELECT * FROM lineitem WHERE l_shipmode = 'MAIL';
2	4	SELECT * FROM lineitem WHERE l_shipmode = 'RAIL';
2	5	SELECT * FROM lineitem WHERE l_shipmode = 'SHIP';
2	6	SELECT * FROM lineitem WHERE l_shipmode = 'TRUCK';
2	7	SELECT * FROM lineitem WHERE l_shipmode = 'AIR' OR l_shipmode = 'FOB';
2	8	SELECT * FROM lineitem WHERE l_shipmode = 'FOB' OR l_shipmode = 'MAIL';
2	9	SELECT * FROM lineitem WHERE l_shipmode = 'MAIL' OR l_shipmode = 'RAIL';
2	10	SELECT * FROM lineitem WHERE l_shipmode = 'RAIL' OR l_shipmode = 'SHIP';
2	11	SELECT * FROM lineitem WHERE l_shipmode = 'SHIP' OR l_shipmode = 'TRUCK';
2	12	SELECT * FROM lineitem WHERE l_shipmode = 'TRUCK' OR l_shipmode = 'AIR';
2	13	SELECT * FROM lineitem WHERE l_orderkey = 1000;
2	14	SELECT * FROM lineitem WHERE l_orderkey > 5000;
2	15	SELECT * FROM lineitem WHERE l_orderkey BETWEEN 200 AND 400;
2	16	SELECT * FROM lineitem WHERE l_orderkey <> 50;
2	17	SELECT * FROM lineitem WHERE l_orderkey IN (10, 20, 30);
2	18	SELECT * FROM lineitem WHERE RIGHT(l_orderkey, 1) = '5';
2	19	SELECT * FROM lineitem WHERE l_orderkey LIKE '%3%';
2	20	SELECT * FROM lineitem WHERE l_orderkey > 10000 AND l_orderkey < 20000;
2	21	SELECT * FROM lineitem WHERE l_orderkey LIKE '1%';
2	22	SELECT * FROM lineitem WHERE l_orderkey IS NULL;

Table A.4: Queries in group 2.

A.4 Tables

Group ID	Query ID	Query Text
3	1	SELECT * FROM lineitem WHERE l_shipmode = 'AIR';
3	2	SELECT * FROM lineitem WHERE l_shipmode = 'FOB';
3	3	SELECT * FROM lineitem WHERE l_shipmode = 'MAIL';
3	4	SELECT * FROM lineitem WHERE l_shipmode = 'RAIL';
3	5	SELECT * FROM lineitem WHERE l_shipmode = 'SHIP';
3	6	SELECT * FROM lineitem WHERE l_shipmode = 'TRUCK';
3	7	SELECT * FROM lineitem WHERE l_shipmode = 'AIR' OR l_shipmode = 'FOB';
3	8	SELECT * FROM lineitem WHERE l_shipmode = 'FOB' OR l_shipmode = 'MAIL';
3	9	SELECT * FROM lineitem WHERE l_shipmode = 'MAIL' OR l_shipmode = 'RAIL';
3	10	SELECT * FROM lineitem WHERE l_shipmode = 'RAIL' OR l_shipmode = 'SHIP';
3	11	SELECT o_orderkey, SUM(l_quantity * l_extendedprice) AS revenue FROM orders o JOIN lineitem l ON o.o_orderkey = l.l_orderkey WHERE l_shipdate BETWEEN '1994-01-01' AND '1994-03-31' GROUP BY o_orderkey ORDER BY revenue DESC;
3	12	SELECT o_orderkey, COUNT(o_orderkey) AS order_count FROM orders o JOIN lineitem l ON o.o_orderkey = l.l_orderkey GROUP BY o_orderkey;
3	13	SELECT * FROM orders o JOIN lineitem l ON o.o_orderkey = l.l_orderkey;

Table A.5: Queries in group 3.