Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Transparent mapping of C++ data frames to database queries

Alice Rey

Masterarbeit im Elitestudiengang Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Transparent mapping of C++ data frames to database queries

Matrikelnummer:	1599247
Beginn der Arbeit:	20. April 2020
Abgabe der Arbeit:	20. Oktober 2020
Erstgutachter:	Prof. Dr. Thomas Neumann
Zweitgutachter:	Prof. Alfons Kemper, Ph.D.
Betreuer:	Moritz Sichert, M. Sc.

SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 20. Oktober 2020 Alice Rey

Abstract

Relational database management systems are state of the art for relational data processing. Especially in-memory databases are well-known for their great performance on datasets of different sizes. Nevertheless, SQL interfaces tend to be unattractive for programmers, since they cannot be used directly in the code. DataFrame abstractions are closely linked to the supporting programming languages and allow the user to run complex queries line-by-line. In SQL, the user has to create one big nested SQL statement and pass it to the database. This way, intermediate results can only be generated by commenting out sections of the query and uncommenting them to return to the whole query.

To overcome the disadvantages of SQL and still be able to work with high-performance relational databases, we present an approach that combines the advantages of data frames with those of relational databases. The user interacts with a DataFrame API implemented in C++. Instead of evaluating the C++ data frames like other data frame supporting frameworks, we map these to SQL database queries. The generated queries are sent to a database for the evaluation. Our work is based on the Spark DataFrame API. Therefore, we evaluate transformations lazily and only trigger the generation of a database query when an action is called.

We evaluate the performance of our data frame mapping with a benchmark that is based on the TPC-H and TPC-DS benchmarks. To apply the benchmark queries to the DataFrame API, we translate them into data frames by chaining Spark's relational transformations. We show that our C++ DataFrame API is at least six times faster when using an in-memory database that works directly on files like Spark. Furthermore, we show that even disk-based databases can achieve better runtimes than Spark when the data is already stored in the database.

Contents

1 Introduction

1.1 Background and Motivation

In the field of relational data processing there exist two types of tools to work with relational data. The classical approach would be to work with a relational database and send requests via a SQL interface to the database. This approach is independent of any programming language which makes it easier to use for non-programmers. As Wu describes in her work [14], the original purpose of SQL is to be used independently through a terminal. Nevertheless, today many programmers need to work with the results generated by a database in their programming environment. They construct the SQL queries as a string in their program and pass them to a database. When working with raw strings instead of programming language constructs, the programmer cannot be supported by the IDE with syntax highlighting or type checking. That makes it difficult for the developer to for example find out where there might be an issue in the query.

The other option for relational data processing is using data frames. A data frame is a data structure and can be utilized more procedurally than the SQL query language. The data structure is implemented in different programming languages and frameworks and is therefore used directly in general-purpose programming languages. It is supported by the programming language R [13] and it is the primary data type of the Pandas framework [9] and the DataFrame API of the Spark framework [1] as well. The Pandas framework is built on top of the Python programming language whereas Spark is available in multiple languages, namely Scala, Python, Java, and R.

Since data frames are used inside of program code, they are first and foremost used by programmers. The idea behind data frames is that programmers can use them directly in a host language, where they can construct the relational queries by invoking a sequence of functions. That is why using data frames instead of the SQL database query language is more procedural. The user can call the operators on the data frame in a sequence one after the other, which is called method chaining. In SQL, the user has to generate one single complex query and cannot easily build up the query in small steps. Thus, he or she is for example not able to check intermediate results. In contrast to SQL, data frames are not represented as raw strings in a host language but are represented as a data structure in the language. Therefore, the programmer can be supported with syntax highlighting and type checking.

SQL APIs are inconvenient to use since constructing a complex SQL query can be very time-consuming. Building a SQL query stepwise or debugging a query is hard since it includes a lot of copying, pasting, and commenting out of some parts to be able to execute only a section of the whole SQL query. Since data frames are built up step by step by calling different operators, debugging is very easy. Each partial result can be stored in a variable. A subsection of operators can be combined to one by creating a function in the host programming language, which makes the operator sequence easier to reuse and less error prone.

Another option that supports the user to interact with a relational database is the object relational mapping (ORM). With the ORM abstraction, programmers can map database tables and their relations to objects in the programming language. The user can use the objects and their class variables to reference columns of data tables in a SQL query. Therefore, the queries are not raw strings anymore, which makes it easier to avoid misspellings in the query. Nevertheless, users often face problems when interacting with ORM abstractions. These problems can occur due to the underlying database schema or the entity classes [7]. In addition, the ORM frameworks are usually used in online-transaction-processing (OLTP) areas for database operations like insertions, deletions or readings. For complex online analytical processing (OLAP) queries the ORM abstraction is less helpful. When the user wants to construct a complex query with ORM, the query is still structured like a standard SQL query. Thus, it still does not enable the user to sequentially define a query like with data frames.

In SQL, the user is encouraged to focus on high-level functionalities. This makes it easy for the relational database to optimize the query. SQL query optimizers decide in which order the sequence of operators is executed [14]. Relational databases are well known for their great performance on any size of relational data, whereas data frames suffer from performance issues already for datasets of medium size [10].

1.2 Problem Statement

Programmers find it more and more inconvenient to use relational databases since SQL, the standard for database management systems, does not support things like syntax-highlighting but instead uses all capital letters due to historical reasons [14]. When SQL was invented, syntax-highlighting did not yet exist. Without syntax-highlighting or any other IDE support, it is hard to understand what a query is doing or to spot invalid parts in an erroneous query. SQL reports error messages after executing a query, which only contain the location of the error. Since a SQL query can combine multiple operators into one query, it is difficult to find out which part of the query causes the error.

Data frames are more user friendly than SQL APIs since they are integrated into programming languages by different frameworks. The code can be split into functions that apply a subset of transformations or actions on a passed data frame and return the resulting data frame. The ability to split the code into smaller pieces makes it easier to produce intermediate results which helps debugging the query plan of the data frame [2].

1.3 Contribution

In this master's thesis we present an approach that combines the benefits of the data frames with those of a relational database. We implemented an API in C++ to work with data frames. As a reference, we used the implementation of the Spark DataFrame API [2] in Scala. We implemented a subset of the available transformations and actions of Spark and made sure that they produce the same results that Spark would produce when called. Apache Spark is a framework for cluster computing, but it can also be used on a single

machine. Databases also support cluster-computing but guaranteeing ACID (atomicity, consistency, isolation, durability) compliance is more complex in a distributed database system than on a single machine. In this master's thesis we focus on the single machine usage.

Differently from Scala, we do not compute the results of actions ourselves. We implemented a flexible backend that can be connected to different databases. Our C++ version of Spark transforms the query plan of the data frames into a database query that is passed to the selected database as soon as an action is called. The main assumption of this master's thesis is that we can improve the runtime of Spark's data frames with a relational database as backend to evaluate the queries.

Since Spark is an in-memory analytics platform [16], we added in addition to the file-based relational database management system a second in-memory database for a better system comparability. As a disk-based database system, we use the open-source system Postgres [11]. Umbra [8] is a high-performance in-memory database with flash-based storage as a fallback in case the dataset is too large to fit into RAM. Besides the standard features of Postgres, Umbra also supports working directly on a CSV file without loading the data into a database table before executing queries. In addition to the in-memory property, directly working on CSV files makes Umbra the most comparable to Spark since the data is not explicitly preprocessed by the database. Since Umbra is a high-performance database, we expect that the in-memory system answers queries faster than Spark. We execute different queries with different data sizes on our implementation in C++ with the two presented databases and compare it to the Spark's shell in Scala. We expect that Umbra achieves better results than Spark since it has better query optimization techniques in addition to the most comparable engine structure.

1.4 Thesis Structure

The thesis is structured as follows: In the following Chapter 2 we explain important terms that are used in the thesis. Chapter 3 explains how we realized the C++ API and which steps have to be performed to map a data frame to a database query. In Chapter 4 our implementation is evaluated with some benchmarks. We evaluate how much time it takes to transform data frames into SQL queries, how well databases can optimize the generated queries, and how much faster our implementation is compared to the Spark shell. Chapter 5 gives an overview of related work and we conclude the thesis with a summary and ideas for future work in Chapter 6.

2 Background

In this chapter, we explain basic terms which we use throughout the work. We start with explaining what Spark is since we use this framework as a base for our implementation. In addition, we compare the performance of our C++ implementation with the one of Spark. Afterwards, we go into more details about the Spark DataFrame API since that is the part of Spark that we are working with. We define what data frames, the core abstraction of the Spark DataFrame API and our C++ implementation, are. In addition, we show how the DataFrame API works and how the user can interact with this API to create data frames. Instead of implementing the same query evaluation functionality as Spark, we work with two different database management systems. At first, we define what a relational database is and secondly, we define what the criteria and characteristics of an in-memory database are.

2.1 Spark

Apache Spark is a cluster computing engine which can be used for multiple purposes like streaming, graph processing or machine learning. The API of Spark is implemented in different languages like Scala, Java and Python [2]. The core abstraction of Spark are Resilient Distributed Datasets, abbreviated RDDs. RDDs are read-only collections. Therefore, when the user invokes transformations like map, filter or reduce, a new RDD is generated. If a RDD is lost, Spark can rebuild it starting from data in reliable store. This is possible since RDDs do not store the data but information about its lineage and all transformations that have been performed to generate the current RDD. Since Spark is mostly working on clusters, the data is partitioned across multiple nodes. If a partition is lost, not all the data is reconstructed but only the data of the failed node. Using this technique, Spark avoids having to replicate all partitions over the network multiple times and the batch computation of the data leads to a higher throughput [12]. Since RDDs do not store the actual data that they represent, they can be evaluated lazily. As soon as an action is invoked, the computation of the dataset is launched. Actions are output operations like for example `count`. Waiting for these operations makes it possible for Spark to optimize the query plan with things like operation pipelining [2]. Even though in general RDDs are computed lazily, the user has the ability to store intermediate results in memory. In addition, the user can optimize the chosen data placement of Spark by controlling the partitioning of the data. If the user for example wants to join RDDs, it makes sense to hash-partition the corresponding data in the same way to guarantee that the corresponding join partners are located on the same nodes [15]. Although RDDs can be optimized due to their lazy evaluation, the optimizations are limited since Spark does not understand the structure of the data of an RDD.

2.1.1 DataFrame API

Spark SQL is an extension of Spark with the goal of supporting relational processing and using DBMS techniques to provide high performance. It offers a programmer friendly API that allows to mix relational analytics with complex analytics by constructing complex pipelines. The entry point for working with the DataFrame API is the `SparkSession`. The Scala Spark Shell already initializes a `SparkSession` object with which the user can interact. To initialize a data frame from a file, the user needs a `DataFrameReader` object. The user can get this by invoking the `read()` method on the `SparkSession` object. With the returned `DataFrameReader` the user can specify properties of the file with the different member functions of `DataFrameReader`. All these functions return a pointer to the same `DataFrameReader` object which enables the user to pipeline multiple function calls. To generate the data frame from the `DataFrameReader` object, the user has to invoke the `load(...)` member function which takes the path to a file as input parameter. In Listing 2.1, we show how a data frame can be initialized in the Scala Spark Shell.

```
var dataFrame = spark.read.format("csv")
    .schema(StructType(List(
      StructField("x", IntegerType, false),
      StructField("y", IntegerType, true),
      StructField("z", StringType, false)
    )
  )).option("delimiter", "|").load("/path/to/file.csv")
```

Listing 2.1: *Initialize a data frame from a file in the Scala Spark shell*

The `schema()` member function takes a `StructType` object as input parameter that contains a list of `StructField` objects. For every `StructField` object, we specify the name, the data type, and if the field is nullable.

Similar to RDDs, data frames are evaluated lazily. The evaluation of a data frame only starts when certain output operations like `count()` or `show()` are invoked. These operations are called "actions". As long as only analyzing methods, so called "transformations", are called, the program returns a new data frame object. In Table 2.1 we listed important transformations and shortly explain their purpose. With this technique the engine is able to perform relational optimization techniques like operation pipelining.

Transformation	**Input Parameter**	**Meaning**
filter(...), where(...)	condition	Filters the rows with the given condition.
select(...)	columns	Selects a set of columns (given as strings or column objects).
withColumnRenamed(...)	column	Renames the given column.
withColumn(...)	column	Adds the given new column or replaces an existing one if it has the same name.
drop(...)	columns	Removes the given column names from the data frame.

join(...)	join partner, optional: Join expression, join type	Joins the data frame with the given join partner, which is also a data frame. As optional parameters, the user can pass a join expression and an explicit join type. Supported join types are “inner”, “outer”, “left-outer”, “right-outer”, “full-outer”, “left-semi” and “left-anti”.
crossJoin(...)	join partner	Cross joins the data frame with the given join partner, synonym to `join(...)` without optional parameters.
distinct()		Returns only unique rows from the data frame.
dropDuplicates()	optional: columns	If no parameter is passed, it is a synonym for `distinct()`, otherwise only the subset of passed columns is considered for duplicates.
limit(...)	number of rows	The resulting data frame contains the first rows of the given data frame, with the number of rows passed as parameter.
sort(...), orderBy(...)	columns	Returns a new data frame which is sorted by the given columns (given as strings or column objects).
as(...), alias(...)	alias	Gives the data frame an alias which can be referenced later.
groupBy(...)	columns	Groups the data frame by the given columns, returns a RelationalGroupedDataset on which aggregate functions can be invoked. Possible functions are listed in Table 2.2.
rollup(...)	columns	Creates a multi-dimensional rollup using the given columns, returns a RelationalGroupedDataset like `groupBy(...)`.
cube(...)	columns	Creates a multi-dimensional cube using the given columns, returns a RelationalGroupedDataset like `groupBy(...)`.

agg(...)	aggregate expressions	Aggregates on the whole data frame without groups.
unionAll(...)	other data frame	Unions the current data frame with the other data frame.
union(...)	other data frame	Synonyme for `unionAll(...)`. For a SQL-style `UNION` that deduplicates the result, the user has to invoke `distinct()` afterwards.
exceptAll(...)	other data frame	Removes the rows from the current data frame that exist in the other data frame. It preserves duplicates.
except(...)	other data frame	Removes the rows from the current data frame that exist in the other data frame.
intersectAll(...)	other data frame	Removes the rows from the current data frame that do not exist in the other data frame. It preserves duplicates.
intersect(...)	other data frame	Removes the rows from the current data frame that do not exist in the other data frame.

Table 2.1: *Transformations of Spark's DataFrame API in Scala*

Aggregates are a special case in the DataFrame API. There exists one function for aggregates that can be directly invoked on a data frame, namely the `agg()` function. In this case the data frame performs the aggregation on the whole set without groups. If the user wants to perform an aggregation on groups, he or she has to invoke the `groupBy(...)` function at first. This function returns a data frame of type `RelationalGroupedDataset`. In Table 2.2 we listed some of the aggregate functions that are offered by the `RelationalGroupedDataset` class. Instead of groups, the user can also invoke the aggregate functions on a multi-dimensional rollup or cube with the corresponding functions which both also return a `RelationalGroupedDataset` object.

In contrast to RDDs, data frames keep track of their schema. Therefore, data frames can be treated like tables in a relational database. At the same time they can be manipulated like RDDs. While RDD optimizations are limited since they do not understand the structure of the data, data frames can support relational operations due to the known schema. This allows executing more optimized evaluation plans. Spark SQL supports the major SQL data types and more complex types such as structs, arrays or maps as well. To be able to execute standard SQL queries, the DataFrame API supports all common relational operators like join, filter, aggregate and project.

Aggregate function	Input Parameter	Meaning
avg(...)	optional: columns	Computes the mean value for all given columns for each group. If no columns are specified, the mean value for all numeric columns for each group is computed.
count()		Counts the number of rows for each group.
max(...)	optional: columns	Computes the maximum value for all given columns for each group. If no columns are specified, the maximum value for all numeric columns for each group is computed.
min(...)	optional: columns	Computes the minimum value for all given columns for each group. If no columns are specified, the minimum value for all numeric columns for each group is computed.
sum(...)	optional: columns	Computes the sum of all values for all given columns for each group. If no columns are specified, the sum of all numeric columns for each group is computed.

Table 2.2: *Aggregate functions of Spark's DataFrame API in Scala that can be invoked on RelationalGroupedDatasets*

In addition to the data frames, there exists another important class in the Spark DataFrame API that is required to invoke some of the relational operators: The column class is used to build complex expressions that are required for example to construct a join or filter condition. Both transformations take a column object as input parameter. The `select(...)` transformation accepts multiple column objects. The most simple column objects are attribute references. In that case, the expression tree of the column object consists of one node. With different column object class functions, there can be constructed more complex expression types. In Table 2.3 we listed some of the column object methods we used. In addition to the column object class functions that can be invoked on existing column objects, there exist additional functions that produce new column objects. Some of these also take column objects as input parameters. We listed these in Table 2.4.

Column object method	Input Parameter	Meaning
asc(), asc_nulls_first(), asc_nulls_last()		Produces a sort expression based on the ascending order of the column. The second method produces a sort expression that returns null values before non-null values and the third method vice versa.
desc(), desc_nulls_first(), desc_nulls_last()		Produces a sort expression based on the descending order of the column. The second method produces a sort expression that returns null values before non-null values and the third method vice versa.
+, plus(...)	other value	Computes the sum of the current column and the given value which can either be a column expression as well or any other value.
-, minus(...)	other value	Subtracts the other value from the current column. As for `plus()` the other value can have any type.
*, multiply(...)	other value	Multiplies the current column by the other value which can have similar to `plus()` any type.
/, divide(...)	other value	Divides the current column by the other value which can have any type like for `plus()`.
%, mod(...)	other value	Returns the current column modulo the other value which again can have any type like for the other arithmetic expressions above.
===, equalTo(...)	other value	Checks the equality of the current column and the other value.
=!=, notEqual(...)	other value	Checks the inequality of the current column and the other value.
<=, leq(...)	other value	Checks if the current column is smaller than or equal to the other value.
>=, geq(...)	other value	Checks if the current column is greater than or equal to the other value.

<, lt(...)	other value	Checks if the current column is smaller than the other value.
>, gt(...)	other value	Checks if the current column is greater than the other value.
&&, and(...)	other value	Boolean AND operator to combine the current column with the other value.
||, or(...)	other value	Boolean OR operator to combine the current column with the other value.
between(...)	lowerBound, upperBound	Returns true if the current column is between the lower and upper bound.
as(...), alias(...), name(...)	alias	Gives the column an alias name.
substr(...)	start position, length	Returns a substring of the current column. The start position and length are specified by the input parameters.
contains(...)	other value	Returns true if the current column contains the other value, based on a string match.
endsWith(...)	other value	Returns true based on a string match if the current column ends with the passed other value.
startsWith(...)	other value	Returns true based on a string match if the current column starts with the passed other value.
like(...)	literal	Returns true if the current column matches the string literal that represents a `LIKE` expression in SQL
rlike(...)	literal	Returns true if the current column matches the string literal that represents a regular expression.
cast(...)	target type	Casts the current column expression to the given target type.
when(...)	condition, value	With this method, one can construct a list of conditions and if a condition is fulfilled, the corresponding value is returned. If no condition is fulfilled and the `otherwise()` function is not used, null is returned.

otherwise(...)	value	The user can call this method after calling the `when()` method one or multiple times. If no condition of the `when()` function is fulfilled, the column object returns the value passed to the `otherwise()` function.
over(...)	optional: window	This method is used to create a windowing function from an analytic function. If no window is passed to the method, the analytic function is evaluated for all rows in the result set. We explain below how windows are created with the DataFrame API.
isin(...)	list of values	The resulting boolean expression evaluates to true if the current column object is contained in the passed list of values.

Table 2.3: *Column object class functions of Spark's DataFrame API in Scala*

Function	**Input Parameter**	**Meaning**
col(...), column(...), $"..."	column name	Converts the given column name into a column object.
lit(...)	literal	Creates a column object of type literal that contains the given literal value.
round(...)	column, optional: scale	Rounds the given column object to the decimal places given by the scale parameter. If the scale parameter is not set, the column object is rounded to zero decimal places.
asc(...), asc_nulls_last(...), asc_nulls_first(...), desc(...), desc_nulls_last(...), desc_nulls_last(...)	column name	Generates a column object from the passed column name and invokes the corresponding column object method.
avg(...), mean(...)	column	Computes the average of the values of the given column.
sum(...)	column	Sums up the values of the given column.
sumDistinct(...)		Sums up the distinct values of the given column.

max(...)	column	Computes the maximum value of the values of the given column.
min(...)	column	Computes the minimum value of the values of the given column.
count(...)	column	Counts the values of the given column.
countDistinct(...)	column	Counts the distinct values of the given column.
when(...)	condition, value	Together with the column object methods `when(...)` and `otherwise(...)`, the user can create list of conditions and corresponding values.
isnull(...)	column	Returns true if the column value is null.
concat(...)	columns	Concatenates all passed column objects.
rank()		This window function is used to return the rank of rows within a window partition.
grouping(...)	column	Indicates if the given column is aggregated or not. This is indicated with 1 if aggregated and 0 otherwise.
abs(...)	column	Computes the absolute value of the given column object.
floor(...)	column	Computes the floor value of the given column object.

Table 2.4: *Additional functions that return column objects in Spark's DataFrame API in Scala*

We can see that the DataFrame API covers lot of expression types such as arithmetic or logical operators with the different functions. If the user wants to create an aggregate expression for the `agg(...)` transformation, he or she can utilize the different aggregation methods like for example `avg()` or `sum()`. As already mentioned, each column object contains an expression tree that consists of multiple expression nodes. These trees are constructed by the user calling different column object functions. An example column object that can be constructed is the following:

```
col("a").substr(2,4) == "abcd" && 3 >= (col("b") + 4) * col("c")
```

In Figure 2.1 we depict the expression tree that represents this column object. The blue boxes contain unresolved attributes that have to be resolved to attribute references. The attribute references are placeholders for columns of the child query plan on which the current transformation is invoked.

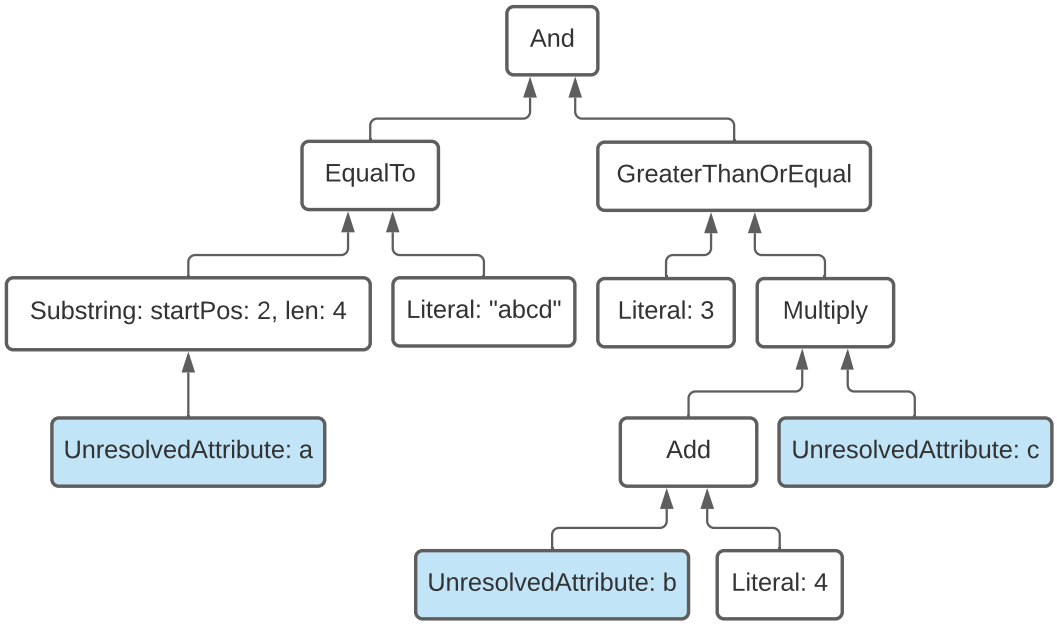


Figure 2.1: *Expression tree for a complex column object*

The `over(...)` function is used to create a window expression. The user passes the window itself to the function as an input parameter. The Window class is a helper class to start constructing an object of type `WindowSpec`. Both the `Window` and `WindowSpec` classes offer the methods listed in Table 2.5. In case of the `Window` class, a new `WindowSpec` is created with either the ordering, the partitioning or the frame boundaries already defined, depending on the called method. If the method is invoked on a `WindowSpec`, a new `WindowSpec` is created. The new `WindowSpec` updates the respective window property of the invoked function with the passed parameter. The other values are set to the values of the current `WindowSpec` object.

2.2 Relational Databases

Relational databases are used for storing and operating on data stored in tabular format. The relational database management system that belongs to the relational database is used to access the data, to guarantee consistency and to modify the stored information. A database management system brings a lot of advantages: It provides solutions for many problems that users have to deal with when handling big amounts of data like uncontrolled redundancy or inconsistency if data is stored multiple times and only one instance is updated. Usually, there exist multiple users that want to access the data stored in a database. In a multiuser system, a database supports organizing access rights and handles anomalies like two users editing the same data in parallel. Database management systems

Window method	Parameter	Meaning
orderBy(...)	columns	The passed column objects specify the ordering of the new window expression.
partitionBy(...)	columns	The passed column objects specify the partitioning of the new window expression.
rangeBetween(...), rowsBetween(...)	start, end	The passed start and end values specify the frame boundaries of the new window expression. There exist three predefined constants: Window.unboundedPreceding, Window.unboundedFollowing and Window.currentRow which are all mapped to special frame boundaries for the resulting window expression.

Table 2.5: *Functions required to construct a Window that can be passed to the over(...) function*

also provide a fail-proof recovery component that enables the system to restore lost data which was written after the last backup copy. In a relational database, the user can specify certain integrity rules that always have to be fulfilled by the stored data. This way, the system rejects operations that do not comply with the integrity rules. Transformations are only executed by the database management system if it does not lead to an inconsistent database state [3].

2.3 In-Memory Databases

In-memory databases keep all data in the main memory. Standard databases swap pages between the buffer memory and the hard disk. Moving the data from hard disk to the main memory is very expensive in terms of runtime. Accessing data from the background memory takes five times longer than accessing data from the main memory [3]. In this master's thesis, we work with Umbra as in-memory database. The Umbra system is not a pure in-memory database, but has evolved from the pure in-memory system HyPer. The HyPer system is a high-performance database system that supports OLTP (online transaction processing) as well as OLAP (online analytical processing). For OLTP transactions it guarantees the ACID properties atomicity, consistency, isolation, and durability [4]. Challenges that in-memory databases face are for example massive parallelism due to multi-core systems. Cache-locality is another point that turns into a bottleneck for in-memory systems since the costs of I/O operations to the background memory, which normally dominate all other costs, do not exist in in-memory systems [3].

Umbra achieves a performance which is comparable to in-memory database systems when working with cached datasets. In addition, it offers scalability like a disk-based system by combining the in-memory buffer with SDDs as storage devices [8].

3 Approach

The goal of this master's thesis is to combine the data frame abstraction with a relational database. This way, we can use the user-friendly API of data frames as our frontend in combination with the query optimization techniques of relational databases. The following chapter explains how we realize data frames in C++ and how we map the C++ data frames to database queries.

The main idea of our approach is to implement the Spark DataFrame API in C++ as similar as possible to the version implemented in Scala. The main difference between the implementations is that Spark evaluates the data frames when an action is called. In C++ we do not evaluate the data frames ourselves. Instead, we map the data frame to a SQL query and send it to a relational database for its evaluation.

We structure this chapter into three sections. In the first section, we explain how data frames can be initialized in our C++ API and how our implementation realizes these. The second section deals with transformations. We explain how query plan graphs are built and what steps are performed by our implementation when a transformation is invoked. The last section deals with the evaluation of the query plan graphs, which is triggered when an action is called. In this step, the query plan graph is transformed into a SQL query that is sent to the database.

3.1 Initializing a data frame in the C++ DataFrame API

In our C++ implementation, the user has two options to generate a new data frame. He or she can either start from a CSV file or a database relation. Since we are working with a database system as backend, we figure that it makes sense to offer an extra option to work with database relations that already exist in the database system.

If the user decides to work with a CSV file, the steps that our implementation performs are similar to the ones performed in Spark. Like in Spark, we work with the `DataFrameReader` class. The user receives a `DataFrameReader` object when calling the `read()` member function of the current `SparkSession` object. We also utilize the different member functions of the `DataFrameReader` class to specify the properties of the CSV file. After the user invokes the `read()` function, a new data frame is created. Each data frame contains a query plan. In this case, the query plan has the type `DataSourceRelation` which is not related to database relations. That is the name Spark uses for this kind of base query plans. The `DataSourceRelation` stores the schema, the source, and the path to the CSV file.

For the second option, we created the new query plan type `DatabaseRelation`. The user can create this type of query plan by invoking the `readFromDatabase()` member function of our `SparkSession` class. Since databases store meta-information about existing tables, the user does not have to specify the schema. Instead, the program uses the meta information of the database to construct the schema itself. In the following we show how a data frame can be created from a database relation:

```
dbFrame = sparkSession->readFromDatabase(<tableSchema>, <tableName>);
```

When the user invokes the readFromDatabase() function, the program requests information about the table from the specified database. In Postgres, we utilize the COLUMNS view of the "information_schema" schema. In the following we show how the corresponding SQL query looks like:

```
SELECT
   column_name,
   data_type,
   is_nullable,
   numeric_precision,
   numeric_scale
FROM information_schema.COLUMNS
WHERE TABLE_NAME = <tableName> AND TABLE_SCHEMA = <tableSchema>
ORDER BY ordinal_position;
```

With the information resulting from this query, our program can create the required StructType object itself. The information_schema.COLUMNS view of Postgres stores information about all columns of all tables currently stored in the database. Since we only need the columns of one table, we filter the rows by their table name and table schema. The StructField objects, that represent the different columns, take the name of the column and its data type as input parameters and a check if the column is nullable. In addition to these three properties, we also select two additional columns, that are required to create a DecimalType object. The constructor of the DecimalType class takes the precision and scale of the contained values as input parameter. In Postgres, these properties are stored in the columns numeric_precision and numeric_scale.

Since Umbra is a research project, it currently does not contain meta-data tables. For this thesis we implemented the information_schema.COLUMNS view to be able to evaluate the performance of DatabaseRelation data frames with Umbra. This is covered in Chapter 4. Whenever a user requests the COLUMNS meta-data view, Umbra scans all tables in the database, retrieves the required information from all columns, and adds a row to the view for each column.

3.2 Analyzing a data frame in the C++ DataFrame API

The member functions of a data frame, which are used to analyze it, are so-called transformations. Like in Spark, these functions create a new data frame instead of modifying the data frame on which the transformation is invoked. Each data frame contains a query plan. In Table 3.1 we listed the different query plan types that are created by the transformations of Table 2.1. A possible chain of transformation calls is:

```
newDf = df.filter(col("x") == col("y")).select(col("x"))
```

In this case the initial data frame "df" contains a DataSourceRelation query plan. The invocation of the filter() transformation creates a new query plan node that references the DataSourceRelation query plan of the data frame "df". The query plan node created by the select() transformation references the Filter query plan.

Query Plan Type	Transformations
Project	select(...), withColumnRenamed(...), withColumn(...), drop(...)
Filter	filter(...), where(...)
Join	join(...), crossJoin(...)
Aggregate	{groupBy(...), cube(...), rollup(...)} + {sum(...), count(...), min(...), max(...), avg(...)}, agg(...)
Distinct	distinct(), dropDuplicates()
Deduplicate	dropDuplicates(...)
Limit	limit(...)
Sort	sort(...), orderBy(...)
Union	union(...), unionAll(...)
Intersect	intersect(...), intersectAll(...)
Except	except(...), exceptAll(...)
SubqueryAlias	as(...), alias(...)

Table 3.1: *Query plan types and their corresponding transformations.*

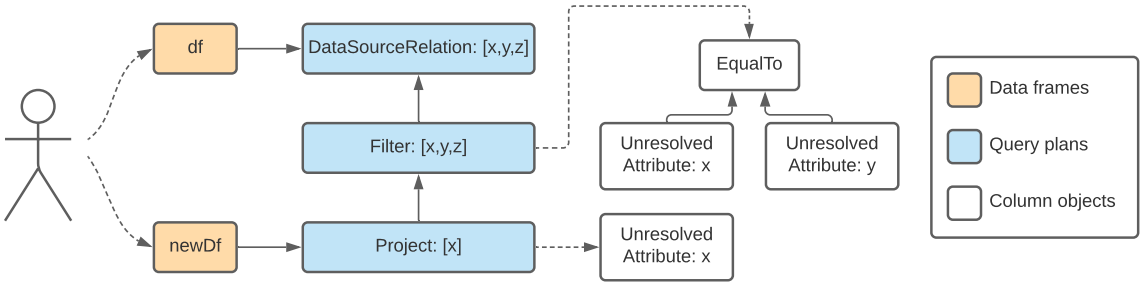


Figure 3.1: *Small query plan graph.*

In Figure 3.1 we depict the query plans in blue boxes with their corresponding data frames drawn in orange boxes. Each new data frame does not reference the previous data frame, but the new query plan contained in the new data frame references the query plan of the previous data frame. Next to the types of the query plan nodes in the blue boxes, we list the columns of the nodes. Our `DataSourceRelation` reads a CSV file with values for the three columns `x`, `y`, and `z`. The `filter()` transformation might remove some rows of the data frame with its filter expression, but the columns do not change. The `select()` transformation only selects column `x`, so the columns change for the corresponding `Project` query plan node.

Besides data frames and query plans, we also work with column objects and expression trees. In our graph in Figure 3.1, some of the transformations take column objects as input parameters. Each column object contains an expression tree. For the `select()` transformation call, visualized by the `Project` query plan node in Figure 3.1, we pass

a column object to express the project expressions. In our case, the project expression consists of one column object that contains a single expression node. The node is an unresolved attribute that is resolved to an attribute referencing the column `x` of the `DataSourceRelation`. The column object we pass to the filter transformation, which is referenced in Figure 3.1 by the `Filter` query plan node, is more complex. The `Filter` query plan stores it as a filter condition. It contains an expression node of type `EqualTo` which has two child expression nodes. The child expression nodes (in white) are two unresolved attributes referencing the columns `x` and `y` of the parent query plan node `Project` in Figure 3.1.

In the following, we explain which steps are performed for each invocation of the different transformation types. Transformation that take column objects as input parameters resolve these unresolved attributes at first. In the example tree in Figure 2.1, all unresolved attributes are drawn in blue boxes. To resolve the unresolved attributes, the current transformation needs to know the columns of the child query plan on which the transformation is invoked. The columns of the child query plans are stored as attribute references. Each attribute reference stores the name of the attribute, the data type, and a check if the column is nullable. Since the user can set an alias for a data frame by invoking the `as()` or `alias()` transformations, the attribute reference also stores a table name and the corresponding table id. This table name and id are updated whenever a `SubqueryAlias` query plan is created.

For most of the query plans, the columns are equal to the corresponding columns of their child queries. For example, if we invoke the `filter()` method on a data frame, we filter out some rows, but we do not change the columns themselves. If someone requests the columns of a query plan of type `Filter`, the query plan requests the columns of its child query plan and returns them as an answer to the request. To avoid unnecessary recomputations, the `Filter` query plan then stores these columns for the following requests. Not only the `Filter` query plan behaves this way, but all other query plans except the `Project` and `Aggregate` query plans reuse the columns of their child query plans. The set operations (`Union`, `Intersect`, and `Except`) combine two child query plans. Since the columns of both child queries are the same, the query plan only requests the columns of one of the child query plans and returns these to the user. In case of a `Join` query plan, the columns of the left and right join partners are concatenated. If the join query plan has type “semi-join” or “anti-join”, only the columns of the left join partner are requested and returned. The `SubqueryAlias` query plan contains the same columns as the child query plan. To allow the user to later reference attributes originating from this query plan with the given alias, we update the table name and id of all columns. The table name is set to the given alias and the id is updated to the current query plan id.

Only for the query plan types `Project` and `Aggregate` the columns change. The `Project` constructor takes multiple column objects as input parameter. These can either be attribute references to columns of the child query plan or more complex expression trees like the one represented in Figure 2.1.

If the column object is just an attribute reference, it is added unmodified to the columns of the `Project` query plan. For more complex expression trees, new attribute references have to be created. We need to retrieve the name, the data type, and the nullability of the

attribute reference from the column object. As table id, we use the id of the `Project` query plan. The table name will be empty until a `SubqueryAlias` transformation is invoked, as already described above. The attribute name is constructed from the expression tree. For our tree in Figure 2.1, the correct name would be:

```
"((substring(a, 2, 4) = abcd) AND (3 >= ((b + 4) * c)))"
```

With this string the user can reference the constructed column in the future. Due to the complexity of the string, he or she can add an alias to the expression, for example by calling the `alias(...)` expression function. In that case, the attribute name is not constructed from all expression tree nodes. Instead, the defined alias name is used.

The data type of the new attribute reference is also retrieved from the expression tree. Starting with the leaf nodes, we check for every following node if the child data types match the operator of the current expression node. Afterwards, we construct the data type of the resulting column object. In case of the expression tree in Figure 2.1, we end up with a Boolean type. For some operators, like the arithmetic operators `Multiply` and `Add`, the resulting numeric type depends on the numeric types of the child query plans.

The nullability of the new attribute reference depends on the nullability of the child query plans. For example, in case of the `And` operator, the resulting expression is nullable if one of the child query plans is nullable. Literals are not nullable, therefore the nullability of the `Add` expression node depends on the nullability of the referenced attribute `b`.

Apart from `select(...)`, there exist other transformations that also generate `Project` query plans. In case of the `withColumnRenamed()` transformation, the program loads all columns of the child query plan. We add a call to the `alias(...)` function to the attribute reference that should be renamed. All other columns of the child query plans are just added to the output of the current query plan. In case of `withColumn()`, we also load all columns from the child query plan. If the name of the new column equals one of the existing columns, this column is replaced with the new column object. Otherwise, the new column object is added as an additional column object. In contrast to adding a new column, the `drop()` transformation removes one of the columns of the child query plan.

The `Aggregate` query plan works with two different lists of expressions: The expressions that define the groups and the aggregate expressions. Both lists are added to the columns of the `Aggregate` query plan. Like for the `Project` query plan, the unresolved attributes of the grouping expression are mapped to the attribute references of the child query plan. The more complex aggregate expressions are treated like the project expressions. The expression tree in Figure 2.1 is no valid aggregate expression since no aggregate operation is invoked. Apart from that, the structure is the same. If the user does not call the `agg(...)` function with column objects that already contain aggregate expressions, the program constructs the aggregate expressions itself. For example, if the `sum(...)` function is called, the program invokes the expression function `sum(...)` on the passed column object, creates a new `Aggregate` query plan node, and passes the constructed expression as aggregate expression. The group definition is taken from the `groupBy(...)` call before the aggregate function call. If the user directly invokes the `agg(...)` transformation, the group's definition stays empty.

After resolving all unresolved attributes and constructing the columns for the new query plan, if necessary, we create a new data frame with the new query plan and return it to the user. Like in Spark, we do not overwrite the data frame on which the transformation is invoked but create a new one.

3.3 Evaluating a data frame in the C++ DataFrame API

Like in the Spark implementation, the evaluation of a data frame is triggered in our implementation by the invocation of an action. In this section, we focus on the actions `show()` and `count()`. As already explained at the beginning of this chapter, in our implementation the `SparkSession` has a pointer to a database object that implements our database interface. For every action this database interface defines a corresponding member function in the database class. Since Umbra and Postgres use the same SQL syntax, we explain the steps for Postgres and only highlight the things that differ in Umbra.

When an action is invoked on a data frame, the data frame calls the corresponding function on the database object that the `SparkSession` points to. At first, Postgres initializes a new `SQLQueryBuilder` object and passes the query plan to its constructor. The `SQLQueryBuilder` retrieves all query plan nodes that are required for the evaluation. Starting from the passed query plan node, the query builder traverses the query plan graph by recursively requesting the child nodes of each query plan. The traversal stops once the leaf nodes are reached. These query plan nodes are either of type `DataSourceRelation` or `DatabaseRelation`. Every query plan node that is reachable from the one query plan node that is referenced by the data frame is stored in a reversely topological sorted list. When an action is invoked on the data frame "`newDf`" in Figure 3.1, the list would start with the `DataSourceRelation` node, continued with the `Filter` node and finish with the `Project` node. If transformations like `union()` are invoked, the query plan graph contains a node with two child query plan nodes. Another special case would be to reuse one query plan node twice, which can be caused by a self join.

After we retrieve all required query plan nodes, we transform the directed acyclic query plan graph into a SQL query. We visit every query plan node separately and create a short common table expression (CTE) for each node.

In the following, we explain what steps are executed by our program for every query plan node. The first nodes that we resolve to SQL queries, are the base nodes which are either of type `DataSourceRelation` or `DatabaseRelation`.

DataSourceRelation. In order to work on a CSV file in Postgres, at first we have to create a new database table and load the data from the CSV file into the database. The information we need to generate a SQL statement can be retrieved from the `StructType` object of the query plan node. In Listing 2.1 we initialize a data frame from a CSV file. The `StructType` object contains three `StructFields` with different data types. In Listing 3.1 we list the required initialization SQL queries for the `DataSourceRelation` query plan node. Since the column names of the CSV file can contain spaces, we enclose these with quotes.

```
-- DataSourceRelation in Postgres
CREATE TABLE table1 (
   "x" int NOT NULL,
   "y" int,
   "z" text NOT NULL
);
COPY table1 FROM '/path/to/file.csv' DELIMITER '|';
...

-- DataSourceRelation in Umbra
WITH table1 AS (
   SELECT *
   FROM umbra.csvview(
      '/path/to/file.csv',
      'DELIMITER ''|''',
      '"x" int NOT NULL, "y" int, "z" text NOT NULL'
   )
)
...

-- DatabaseRelation
WITH table1 AS (
   SELECT *
   FROM <tableName>
)
...
```

Listing 3.1: *Example SQL statements for DataSourceRelation and DatabaseRelation query plans*

Umbra allows us to work directly on a CSV file. Thus, we would create a CTE for the `DataSourceRelation` query plan of Listing 2.1. The second query in Listing 3.1 shows how we deal with CSV files in Umbra. We do not need any initialization queries for reading the file. Therefore, we just add a CTE query that can be referenced by the following queries.

DatabaseRelation. A `DatabaseRelation` query plan does not require any mappings. The generated CTE in this case is identical for Umbra and Postgres. We select all columns from the requested table name such that following query plans can reference them utilizing the CTE's alias. The third query in Listing 3.1 shows how the `DatabaseRelation` query plan is expressed with SQL.

Filter. After we visit all base nodes, we start visiting nodes that reference one or two child query plans. In our example (see Figure 3.1), the next query plan node has the type filter. In SQL, a filter query plan is mapped to a query that selects all columns from the child query plan. The CTE contains a `WHERE`-clause in which the column object is mapped to a database expression. In Listing 3.2 we show the corresponding SQL CTE.

Project. The last query plan we visit is the `Project` query plan. Since we reach the query plan that is referenced by the data frame on which the action is invoked, we do not construct another CTE, but our main query which we can also see in Listing 3.2. In this

```
-- Filter
WITH filter2 AS (
   SELECT *
   FROM table1
   WHERE x = y
)

-- PROJECT
SELECT x
FROM filter2;
```

Listing 3.2: *Example SQL statement for the Filter and Project query plans*

```
WITH table1 AS (
SELECT *
FROM umbra.csvview(
'/path/to/file.csv',
'DELIMITER ''|''',
'"x" int NOT NULL, "y" int, "z" text NOT NULL'
)
),
filter2 AS (
SELECT *
FROM table1
WHERE x = y
),
project3 AS (
SELECT x
FROM filter2
)
SELECT COUNT(*) FROM project3;
```

Listing 3.3: *Example SQL statement for the count() action*

example query, the project expression list only contains a simple attribute reference. In Spark, newly constructed columns are still accessible in the following transformation calls. To guarantee this property in our implementation as well, we add to the more complex column objects their attribute names as an alias. In the previous section, we already showed how the attribute name for the column object of Figure 2.1 would look like.

When combining one of the queries of Listing 3.1 with the query of Listing 3.2 we get the complete query our program would generate for the data frame `newDf` when the action `show()` is invoked.

In case of a `count()` invocation, the corresponding function in the database class adds another query plan node to the query plan graph of type `Aggregate` that counts all rows of the query plan. Afterwards, the function passes the modified query plan graph to the `SQLQueryBuilder` for the evaluation. In our resulting SQL query which we show in Listing 3.3, the `PROJECT` query is a CTE as well and our main query counts all rows. The corresponding data frame query looks as follows:

```
df.filter(col("x") == col("y")).select(col("x")).count()
```

In Table 3.2 we list the resulting database queries for all query plan types of Table 3.1. The second column contains the required parameters, whereas the third column shows the common table expression with placeholders for the respective parameters. Query plan types with one child node have the additional parameter `<childNode>`. Query plans with two child nodes, like the `Join` or `Union` query plans, have the two additional parameters `<leftChildNode>` and `<rightChildNode>`.

Query Plan Type	**Parameter**	**SQL Query**
Project	projectExpressions	`SELECT <projectExpression>` `FROM <childNode>`

Filter	filterExpression	`SELECT *`
		`FROM <childNode>`
		`WHERE <filterExpression>`

Cross Join		`SELECT *`
		`FROM <leftChildNode>`
		`  CROSS JOIN <rightChildNode>`

Inner Join (similar for outer join)	joinExpression	`SELECT *`
		`FROM <leftChildNode>`
		`  INNER JOIN <rightChildNode>`
		`  ON <joinExpression>`

Semi Join	joinExpression	`SELECT *`
		`FROM <leftChildNode>`
		`WHERE EXSITS (`
		`  SELECT *`
		`  FROM <rightChildNode>`
		`  WHERE <joinExpression>`
		`)`

Anti Join	joinExpression	`SELECT *`
		`FROM <leftChildNode>`
		`WHERE NOT EXSITS (`
		`  SELECT *`
		`  FROM <rightChildNode>`
		`  WHERE <joinExpression>`
		`)`

Aggregate	groupExpressions, aggregate-Expressions	`SELECT <groupExpressions>,`
		`  <aggregateExpressions>`
		`FROM <childNode>`
		`GROUP BY <groupExpressions>`

Distinct		SELECT DISTINCT * FROM <childNode>
Deduplicate	attributes	SELECT DISTINCT ON(<attributes>) * FROM <childNode>
Limit	number	SELECT * FROM <childNode> LIMIT <number>
Sort	sortExpressions	SELECT * FROM <childNode> ORDER BY <sortExpressions>
Union (same for union all)		SELECT * FROM <leftChildNode> UNION ALL SELECT * FROM <rightChildNode>
Intersect (similar for except)		SELECT * FROM <leftChildNode> INTERSECT SELECT * FROM <rightChildNode>
Intersect All (similar for except all)		SELECT * FROM <leftChildNode> INTERSECT ALL SELECT * FROM <rightChildNode>
SubqueryAlias	name	WITH <name> AS (SELECT * FROM <childNode>)

Table 3.2: *CTEs for the different query plan types listed in Table 3.1*

For most of the query plan types the mapping is straightforward. We split the `Join` query plan into four rows since the different join types that Spark supports result in different kind of queries. The cross join is the only one where no expression is passed. For the inner join and the different outer joins the resulting queries are similar. All join types contain a join condition and the only thing that differs is the respective `JOIN` keyword. The semi join produces a correlated subquery where we make use of an `EXISTS` clause to check if the right join partner contains any row that fulfills the join condition. For the anti join, we utilize a `NOT EXISTS` clause at the same place.

The `Aggregate` query plan contains two expression lists: One list is passed to the program with the `groupBy()` transformation and the other one is generated from the used

```
-- Incorrect: Separate Sort and Limit query plans
WITH sort1 AS (
   SELECT *
   FROM a
   ORDER BY a, b
),
limit2 AS (
        SELECT *
        FROM sort1
        LIMIT 2
)
...

-- Correct: One combined query plan for Sort and Limit
WITH sort1 AS (
   SELECT *
   FROM a
   ORDER BY a, b
   LIMIT 2
)
...
```

Listing 3.4: *Example SQL statements for the sort and following limit query plan*

aggregate transformation. Both lists are added to the `SELECT` clause, starting with the group expressions and followed by the aggregate expressions. In the `GROUP BY` clause only the group expressions are inserted.

The result set of a `Limit` or `Deduplicate` query plan depends on the used sorting criteria in SQL. Especially Umbra does not track any previous sorting criteria and even skips all unnecessary sort expressions. In addition, the sorting criteria is only taken into consideration if it is part of the CTE for the `Limit` and `Deduplicate` query plans. Therefore, we have to combine the `Sort` query plan node with query plans like `Limit` or `Deduplicate`. In Listing 3.4 the first version shows how the SQL query would look with separate subqueries. This version is incorrect. The second version shows how the combined correct query looks like for the data frame `a.sort("x","y").limit(2)`. In our implementation, we only check for the directly referenced child node if it has the type `Sort`. In Spark, the user is also able to define the sort criteria for other previous query plan nodes. For our purposes checking only the directly referenced child is sufficient. For an overall semantic correctness, every query plan node has to track the current sorting criteria. That way, the current sorting criteria can be applied whenever required.

Another special case is the `Union` query plan. In Spark, the `union()` transformation is a synonym for the `unionAll()` function. In SQL, a `UNION` operator produces a result set that does not contain any duplicates. Whereas in Spark, the user has to invoke the `distinct()` transformation explicitly after calling the `union()` or the `unionAll()` transformation. Since we use Spark as our template and try to mimic the behavior of Spark with our version, we map both transformations to the `UNION ALL` operator in SQL.

The `SubqueryAlias` query plan, which is the last entry in the Table 3.2, is the only one where we display the entire `WITH` query statement. The query definition selects all

columns from the child node. The name of the CTE is defined by the input parameter. The most important thing that happens when a `SubqueryAlias` query plan is invoked is that the table names of the attribute references are updated. We already mentioned that in the previous section since these steps are already performed when the transformation is invoked. We use the name of the query as the name of the CTE, but since in this case the alias could only be accessed by the directly following query plan, we need to store the alias as the table name in the attribute reference objects as well. If the user later tries to use one of the columns with the defined table alias, the program searches for the attribute reference by name and alias table name. As an example, that gets handy for a self join where the user assigns two different aliases to the child node to be able to tell the left and right join partner apart. The only thing that makes it possible for the program to tell the columns of the resulting join plan apart is the table names.

The `SQLQueryBuilder` visits the query plan nodes in reverse topological order. For each node, at first we load the reference to the child queries. In addition to the name of the child CTEs, we also load the names of the columns that are used internally by our SQL statement. There are some cases where the internal names of the columns differ from the ones that are used in the DataFrame API.

The reason for this internal renaming is the `Join` query plan. As shown in Table 3.2, we use the star operator to select all columns of the left and right child partners. In practice at first we have to make sure that the left and right join partners do not contain columns with the same name. Otherwise, our SQL query would not be able to tell them apart in following CTEs. The data frame query `a.join(b, a("x") == b("x")).select(b("x"))` is able to select the column `x` of the right join partner `b` after the `join()` transformation. If we construct the query like presented in Table 3.2, we would end up with the first SQL query in Listing 3.5. This query fails since the column `x` is ambiguous. The second query shows what we do in our implementation to deal with duplicate column names. When we notice duplicate column names, we rename the columns of the right join partner. We concatenate the table name with the column name. In the following `PROJECT` query plan node, the column object `b("x")` references a column of the previously right join partner `b`. To be able to retrieve the renaming, we store which columns we renamed for every pair of query plan node and child query plan node. Since the user later references a column of the query plan node `b` in Listing 3.5, we check if the `Join` query plan renamed the column `x` of the child node `b` and end up with the current internal name `b_x`.

After resolving all attributes to the internal column names, we generate the CTE for the current query plan node. Afterwards, we store the new internal column names of the generated query for the following query plan node.

Until this point, we treated all query plan nodes the same and always generated a CTE. If a query plan node is only referenced by one other query plan as a child node, we do not have to generate a CTE. Instead, we can directly paste the query as subquery into the `FROM` clause of the parent query.

The data frame query `df.filter("x"_c < "y"_c).select("x"_c).show()` is a candidate for using subqueries instead of CTEs since the `Filter` node is only utilized by the

```
-- Join Query without renaming
WITH join1 AS (
   SELECT *
   FROM a,b
   WHERE a.x = b.x
)
SELECT x FROM join1;

-- Join Query with renaming
WITH join1 AS (
   SELECT a.*, b.x as b_x, b.y as b_y, b.z as b_z
   FROM a,b
   WHERE a.x = b.x
)
SELECT b_x FROM join1;
```

Listing 3.5: *Example SQL statements for the join query plan*

```
-- Query with CTEs
WITH filter1 AS (
   SELECT *
   FROM df
   WHERE x < y
)
SELECT x FROM join1;

-- Query with subqueries
SELECT x
FROM (SELECT *
      FROM df
      WHERE x < y) filter1;
```

Listing 3.6: *Example SQL queries with CTE or using a subquery*

`Project` node. In Listing 3.6, we display both the CTE and the subquery version.

In our implementation, we always use the subquery version if possible. If a query plan node is visited, we check how many other nodes use the current node as a child query plan before we generate the query . If there is only one referencing node, we create a subquery and add an alias to it. If more than one node is referencing the query plan, we still generate a CTE. All referencing queries will have to utilize the CTE name in their `FROM` clause.

In case of one referencing query plan node, we do not copy the string of the subquery into our new query since this would lead to many unnecessary copy operations. Instead, we build up our query with a doubly linked list of string chunks. In Figure 3.2, we demonstrate how this list looks like for a `Join` query plan with two child query plan nodes. The `Join` query chunks, drawn in blue boxes, are added in front of the first child, between the child nodes, and behind the second child node. The child nodes are drawn in orange boxes. A following query plan that references the `Join` query plan, would add its chunks to the front and back of the string chunk list. In this way we add new chunks until we either reach a node that is referenced multiple times or the main query node. If a node is reached that is

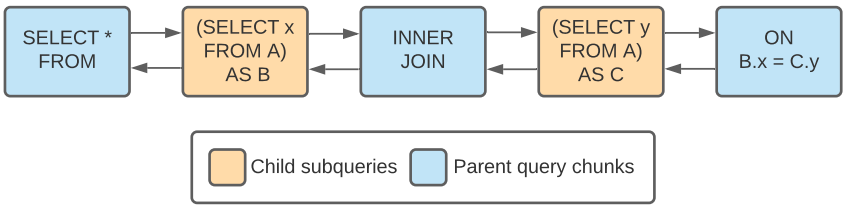


Figure 3.2: *Doubly linked list of SQL query with child queries*

referenced multiple times, we generate a CTE for it and add the concatenated subquery chunks to its `FROM` clause. The following referencing query nodes start generating a new chunk list with the reference name of the CTE instead of the whole subquery string. If the main query node is reached, the chunks are concatenated, and the main query is generated. The main query is concatenated with the initialization queries and the CTEs and is sent to the database.

In Table 3.2, some of the query plan types have column object parameters which we do not resolve in the SQL query. In Table 3.3, we list the different column object functions and show to which SQL expression these functions are mapped. Functions which are treated similar in SQL, are combined into one row. For most of the functions the mapping is straightforward since expressing them in SQL is sometimes even identical to the format in Spark.

A special case are the string functions `contains()`, `endsWith()` and `startsWith()`. We use a `LIKE` operator in SQL to mimic this behavior. In case of `contains()`, we concatenate the expression, that should be contained, with a percent sign before and after the given value, respectively, which allows for arbitrary additional characters surrounding the column object `b`. In case of the `endsWith()` function, we only add a percent sign at the front and for the `startsWith()` function only at the back.

The `substring()` function needs some additional SQL functions to work as expected. In SQL, the `SUBSTRING` function can only be invoked on string columns. Since Spark also supports the function for other data types, we cast the passed column object to a `varchar`. The position argument is wrapped in our mapping into a `CASE WHEN` operator. The reason for that is, that in Spark, the position "0" is a valid argument and is identical to the position "1". In SQL position "0" is not supported and therefore we map the value to position "1" explicitly if it equals "0".

Window expressions are the most complex since in Spark the user passes a `Window` object to the `over()` function that contains information about the partitioning, the ordering, and the frame boundaries. Mapping the partitioning and ordering is straightforward. The only difference between Spark and SQL is the definition of the boundaries. In Spark, the user passes two values to the `rowsBetween()` function which can be negative or positive. A negative value means that the user references a preceding row and a positive value means that he or she references a following row. In SQL, only positive values are supported. To

reference a preceding row in SQL, the `PRECEDING` keyword is used. If the value is negative, we compute its absolute value and use the `PRECEDING` keyword. Otherwise, we use the `FOLLOWING` keyword.

Column object expression	**Expression in SQL**
`abs(a), floor(b), grouping(c), not(d)`	`ABS(a), FLOOR(b), GROUPING(c), NOT(d)`
`a + b, c / d, e * f, g % h, i - k`	`a + b, c / d, e * f, g % h, i - k`
`avg(a), min(b), max(c), count(d), sum(e)`	`AVG(a), MIN(b), MAX(c), COUNT(d), SUM(e)`
`a.as("name")`	`a AS "name"`
`a && b, c || d`	`a AND b, c OR d`
`a == b, c != d, e <= f, g < h, i >= k , m > n`	`a = b, c != d, e <= f, g < h, i >= k, m > n`
`when(a,b).otherwise(e)`	`CASE WHEN a THEN b ELSE c END`
`cast(a,IntegerType())`	`CAST(a AS int)`
`concat(a,b,c)`	`a || b || c`
`a.contains(b)`	`a LIKE '%' || b || '%'`
`a.endsWith(b)`	`a LIKE '%' || b`
`a.startsWith(b)`	`a LIKE b || '%'`
`a.in(b,c)`	`a IN (b,c)`
`a.isnull()`	`a IS NULL`
`a.like(b)`	`a LIKE b`
`rank()`	`RANK()`
`a.rlike(b)`	`CAST(a AS VARCHAR) SIMILAR TO b`
`a.round(x)`	`ROUND(a,x)`
`a.subString(pos,len)`	`SUBSTRING( CAST(a AS VARCHAR), (CASE WHEN pos = 0 THEN 1 ELSE pos END), len )`
`avg(a).over( Window .partitionBy(b) .orderBy(c) .rowsBetween(-x,y) )`	`AVG(a) OVER ( PARTITION BY b ORDER BY c ROWS BETWEEN x PRECEDING AND y FOLLOWING )`

]

Table 3.3: *Mapping of column object functions of Table 2.3 and Table 2.4 to SQL*

4 Evaluation

In this chapter, we want to measure the performance of our C++ implementation of the Spark API which uses a database relation to solve the data frame queries. Therefore, we compare the Scala Spark implementation with our C++ implementation. As relational database management systems which are solving the queries, we use Postgres and Umbra. Additionally, we evaluate the overhead of the C++ DataFrame API compared to directly using a SQL interface. We measure the time our implementation needs to map the data frames to database queries, and we check how well the used databases can optimize the constructed database queries.

4.1 Research Questions

In this master's thesis, the following research questions are proposed, and their results are presented and discussed.

RQ1: How fast can a relational database answer data frame queries? With the first research question, we want to evaluate how fast Postgres and Umbra can evaluate a data frame query compared to Spark. We compare how the databases perform when the data is already stored in the database and how the databases perform when the tables first have to be created before executing the different queries.

RQ2: How fast can a relational database working with CSV table scans answer data frame queries? In contrast to Spark, databases preprocess the data when a new table is created. If we want to get the first few rows of a data frame, Spark only reads the first lines from the specified data source. On the contrary, a database loads the entire data set into the database before returning the first few lines. In Umbra, there is the possibility to work directly on CSV files, which is more similar to how Spark works with data. We compare the runtime of different queries in Umbra when utilizing this `csvview` function with Spark.

RQ3: How time-consuming is the usage of the C++ DataFrame API instead of a SQL interface? Working with a DataFrame API instead of a SQL interface can have usability benefits, but how much time does the mapping need? With some microbenchmarks we evaluate some corner cases of the mapping and check how much time it takes to map the data frames to database queries.

RQ4: How well can a relational database optimize queries generated from data frames? For query optimizations we rely on the optimizations that are done by the utilized databases. Our constructed SQL queries differ from those a user would write when directly interacting with a SQL interface. With this research question we evaluate how well databases can optimize our constructed SQL queries compared to queries directly

formulated in SQL.

4.2 Study Objects

As the first study object, we use the dataset and queries of the TPC-H benchmark. TPC-H is a decision support benchmark with an industry-wide relevance. To be able to utilize the queries with our Spark API, we converted the SQL queries to chained method calls to the DataFrame API. In Appendix A we show for query TPC-H 4 how the SQL query looks like when it is transformed to the DataFrame API.

Since no TPC-H benchmark query contains a window function, we use as a second study object the dataset and some queries of TPC-DS as well. TPC-DS is also a decision support benchmark and provides a representative evaluation of general-purpose decision support systems.

The relations of TPC-H and TPC-DS all contain primary keys. For our evaluation, we remove the primary keys option from the `CREATE TABLE` statements. Spark does not take the primary key as an optional parameter, so if we would use this extra information in the databases, Spark would be disadvantaged.

For the third research question, we work with additional microbenchmark queries to be able to test the performance of our C++ DataFrame API for some edge cases. We created five data frame queries. Four queries repeatedly execute the same operator on a theoretical data source relation which does not have to exist since we only measure the time it takes to construct the SQL query. We do not measure the time it would take to evaluate the query on a relational database. The operators we evaluate are `FILTER`, `AGGREGATE`, `UNION` and `JOIN`. The fifth microbenchmark evaluates the column object type of the Spark DataFrame API which is used to build expressions. In this scenario, we only call the filter operator once, but we vary the size of the filter expression. The meaning of the resulting SQL queries is not relevant in this benchmark, since we do not pass the generated SQL queries to a database system to evaluate them. It is just important that the queries are syntactically correct.

For the last research question, we reuse the TPC-H queries from the first two research questions to compare the generated SQL queries with the standard SQL queries. We retrieve the generated SQL queries from our C++ DataFrame API implementation where we normally generate the SQL queries to send them to a given database. We capture these requests and then manually send these SQL queries to the Postgres and Umbra databases. In Appendix B we show how the generated SQL query looks like for TPC-H query 4.

4.3 How fast can a relational database answer data frame queries?

4.3.1 Study Design

For the first research question, we compare the runtimes of Spark with the runtimes of our C++ DataFrame API with different databases. As standard open-source database we use

Postgres and as in-memory database we use Umbra. For both databases, we work with two different setups. For the first setup, we measure the execution time of our C++ DataFrame API when working with `DataSourceRelations`. In that case, Postgres and Umbra have to load the required tables into the database first before the actual query can be executed. The time it takes to create the tables and to copy all the required data into the tables, is a part of the query execution time. In the second setup, we load the TPC-H relations into the database before executing the queries. Therefore, we can work in our DataFrame API with `DatabaseRelations` as base relations. The time it takes to preload the data is not added to the execution time of the query in this case.

In addition to the different base relations, we also use different data sizes of the TPC-H and TPC-DS datasets. For Postgres we use the scale factors 1, 5, and 10. These correspond to the dataset sizes 1 GB, 5 GB, and 10 GB. For Umbra we use scale factor 20 in addition to the scale factors already used for Postgres. We repeat each query five times for each setup and report for every combination the fastest execution time. To compare the execution times with Spark, we execute the same queries in the Spark Shell of Scala as well. We restart the Spark Session after every repetition to avoid that Spark keeps any precomputations that would accelerate subsequent computations.

All scenarios of all four research questions are performed on an Intel Xeon E5-2667v4 CPU with 8 physical and 16 logical cores running at 3.2 GHz. The system contains 128 GB of main memory (8x 16 GB DDR4-2400 ECC memory modules) and we run Ubuntu 20.04.1 LTS on it. For our data, we use the RAM file system 'ramfs' of Linux to get rid of the reading and writing overhead from an SSD. We work with Spark 3.1 and PostgreSQL 12.4. Before any benchmark is executed, we store the TPC-H data of the required scale factor in a subdirectory there. For Postgres, Umbra, and Spark we create additional subdirectories and link the engines to these. The engines will use the directories as storage locations.

In addition to TPC-H, we also evaluate the performance of two TPC-DS queries in this research question. The queries we selected from TPC-DS both contain a window function since no query of TPC-H covers these. We choose the queries 12 and 64 and run them on different dataset sizes from 1 GB to 20 GB. We are only able to execute the TPC-DS queries in Postgres because Umbra does not yet support window functions.

In Spark the action `show()` outputs a string representation of the first 20 result rows. In C++ we implemented the `show()` function as well and therefore compute the same string representation as Spark. For our evaluation we compare the runtimes of the Spark Scala implementation and our C++ version. Generating the string representation and outputting it to the not-existing user is not relevant for the benchmark, so we added an additional function that skips the string generation.The additional function, named `benchmark()`, performs the same steps like the `show()` function without the string computation. Instead, the function returns an array of resulting rows. In addition, we do not limit the resulting rows but compute all rows by setting the `rowCount` variable to the maximum array length.

4.3.2 Results

In Figure 4.1, Figure 4.2, Figure 4.3, and Figure 4.4, we display the relative speedup of the different database setups and the different scale factors compared to Spark. In the graphs

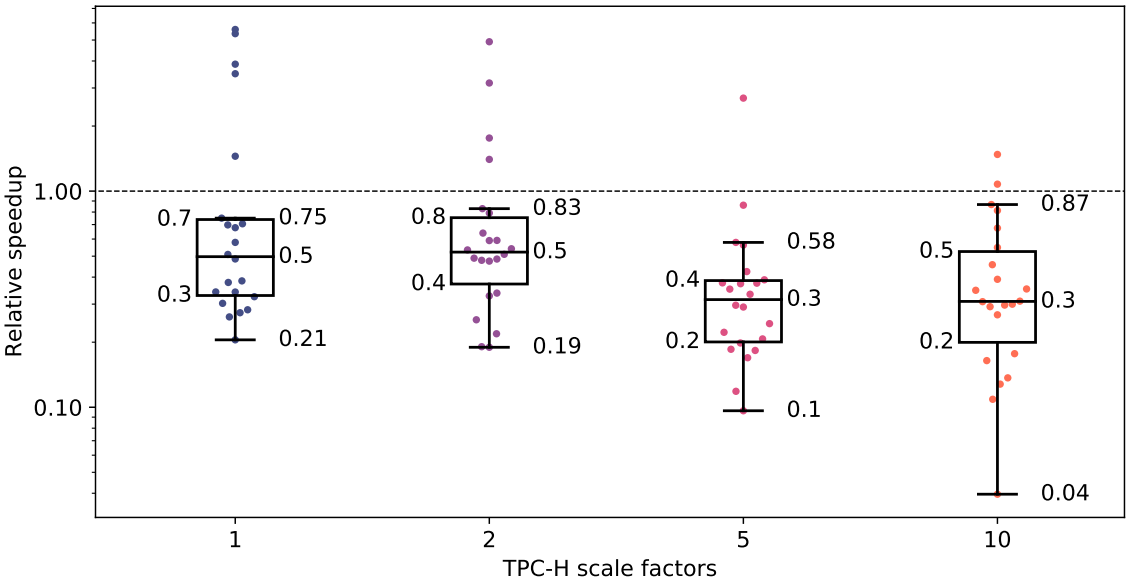


Figure 4.1: *Relative speedup of C++ DataFrame API with Postgres (data loaded from CSV files) over Spark DataFrame API*

we draw a horizontal line at the relative speedup of 1 to highlight the border between a actual speedup and a slowdown of the relational databases. That means, if a value is underneath the border, the Spark version is faster than our C++ version. On the left side of the boxplots, we documented the value of the first and third quartile and on the right side in the middle, we wrote the median value. Below and above the median value on the right side, the values of the lower and upper whiskers are written. Data points which are plotted above and underneath these whiskers are considered outliers.

We start with the relational database Postgres and later take a look at the figures that present the results for the in-memory database Umbra. Figure 4.1 contains the execution time of Postgres when we work with `DataSourceRelations`. In that case, Postgres at first has to create all required tables in the database and then has to copy all the data into the tables. Afterwards, the engine can start to compute the actual SQL query. We observe that in this scenario Postgres needs mostly more time to execute the TPC-H queries than Spark. The median speedup for all scale factors is smaller than or equal to 0.5, which means that Spark is at least two times faster than Postgres for the median value. There are only a few outliers for each scale factor where the calculation of Postgres is faster. For all four scale factors, the queries 2 and 11 are evaluated faster in Postgres than in Spark. In addition to these two, the queries 13 and 16 are evaluated faster in Postgres than in Spark for the scale factors 1 and 2. When comparing the boxplots in this figure, we can see that for the scale factors 1 and 2 the speedups of Postgres are similar. The

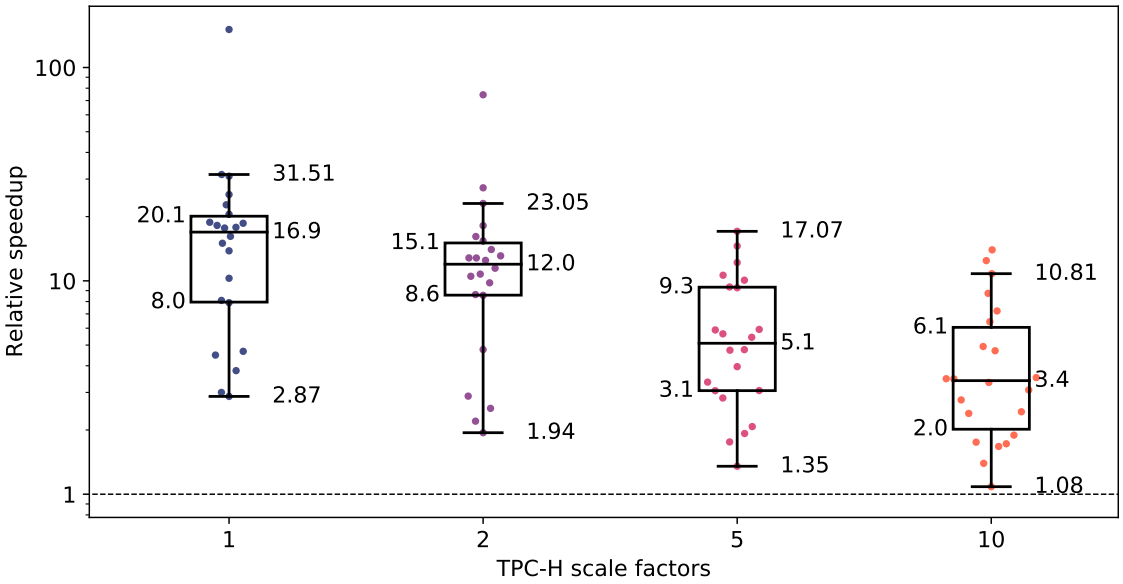


Figure 4.2: *Relative speedup of C++ DataFrame API with Postgres (data stored in the database) over Spark DataFrame API*

medians for these scale factors are the same. The only difference is that for scale factor 2 Postgres has a slightly higher dispersion of the speedups. For scale factors 5 and 10 the medians are identical as well, but this time the dispersion is significantly higher for scale factor 10 than for scale factor 5. In the right most boxplot we observe that one query is significantly slower when evaluated in Postgres compared to the others. This query has the query number 7 and is executed more than 25 times faster in Spark than in Postgres.

In Figure 4.2 we see the results of Postgres when we work with `DatabaseRelations`. This time, the data is already available in the database. We observe that Postgres executes in that case all queries faster than Spark. Nevertheless, Spark gets closer to the execution times of Postgres with a growing data size. For a smaller data size of 1 GB, Postgres has a median relative speedup of 16.9. The median shrinks with a growing scale factor and for scale factor 10, Postgres is only 3.4 times faster than Spark for the median value. This is still high, but compared to scale factor 1 we observe a speedup loss. For all four scale factors, the queries for which Postgres achieves the highest relative speedup are the queries 11 and 15. On the other side, especially for scale factor 10 there exist some queries that are not even twice as fast in Postgres than in Spark, even though Postgres already loaded the data into the database. For example, query 20 is more than four times faster for scale factor 1 but has the smallest speedup for scale factor 20 with a speedup factor of 1.08.

Umbra is able to produce higher speedups for both the `DataSourceRelations` and the

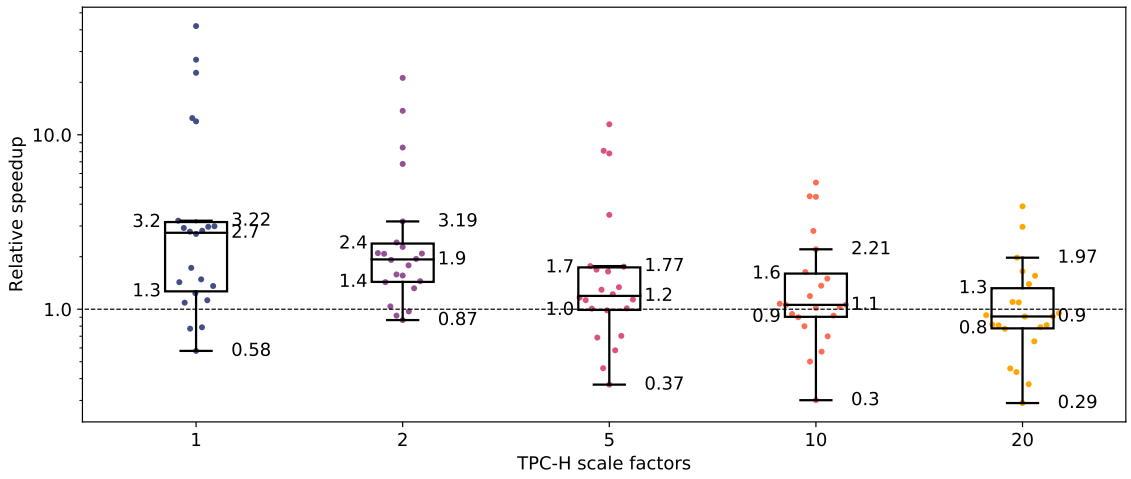


Figure 4.3: *Relative speedup of C++ DataFrame API with Umbra (data loaded from CSV files) over Spark DataFrame API*

`DatabaseRelations` compared to Postgres. For the Umbra scenarios we also compute the runtimes for scale factor 20 in addition to the scale factors from 1 to 10 . In Figure 4.3 we see that there exist some queries which are faster in Spark than in Umbra. However, in contrast to Postgres, the median relative speedup is bigger than one for all four scale factors. Like in the Postgres scenarios, the median speedup shrinks with a growing data size. Queries 2, 11, and 16 are outliers of the first four scale factors. In case of scale factor 20, only queries 2 and 11 are outliers of the boxplot. The other outlier queries vary depending on the scale factors. Some of the queries are slower when executed with Umbra. Especially with a higher scale factor, the number of slower queries and the slowdown get higher. In case of scale factor 10, eight queries are slower when using Umbra and the C++ API instead of the Spark DataFrame API of Scala. For scale factor 20, more than half of the queries (14) are slower in Umbra. For the queries 18 to 21, Spark is more than two times faster than Umbra for scale factor 20. For scale factor 10, Spark executes query 21 two times faster than Umbra and query 14 takes more than three times longer in Umbra than in Spark. For scale factor 5, only six queries are slower and in the case of scale factors 1 and 2, only three queries need more time with Umbra than Spark. With a relative speedup of 0.37 query 14 is also the slowest compared to Spark for scale factor 5.

The best results for Umbra can be observed in Figure 4.4 where the tables are already available in the database and being accessed with `DatabaseRelation` data frames. In this scenario, all queries are significantly faster with Umbra than with Spark. For scale factor 1 all queries are at least 85 times faster and for scale factor 20, they are still at least 27 times faster. We observe a decline of the lower whisker of the box plots like for all other database setups. In contrast to the previous scenario of Umbra, the median value only gets smaller for the first four scale factors. For scale factor 20, the median value gets again

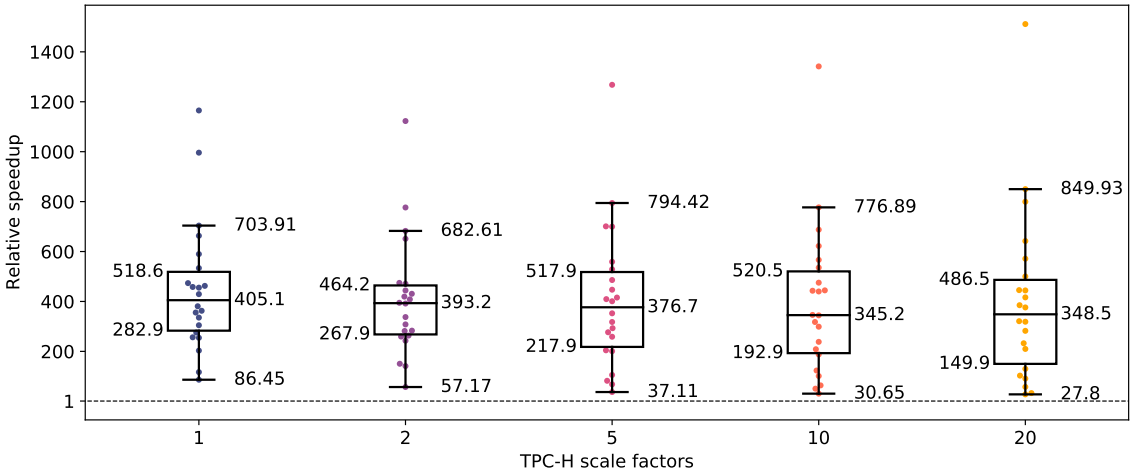
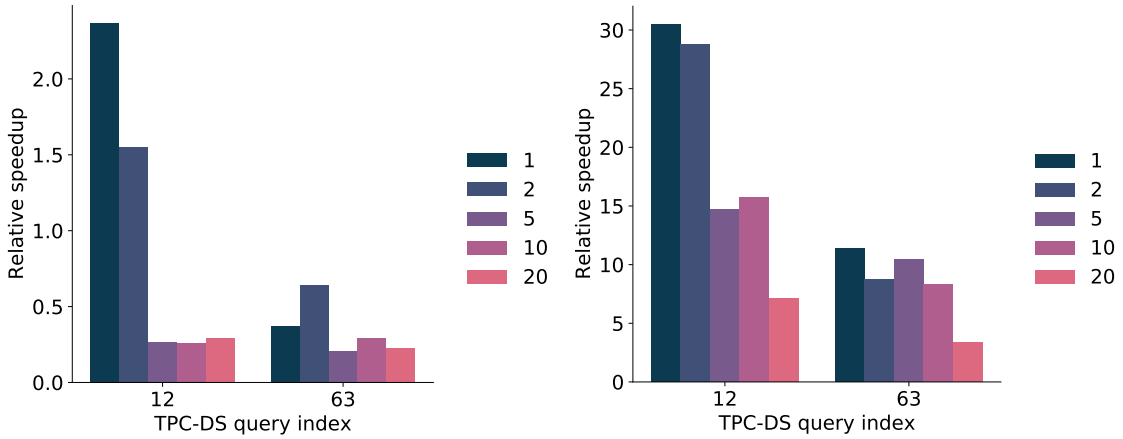


Figure 4.4: *Relative speedup of the C++ DataFrame API with Umbra (data stored in the database) over the Spark DataFrame API*

bigger than for the previous scale factor 10.

Like for the other scenarios, there exist some outliers for all four scale factors. Query 15 is an outlier for all five used scale factors and in contrast to the other queries, the speedup is growing with a growing data size. For scale factor 1, query 15 is about 1000 times faster when executed with Umbra and for scale factor 20, Umbra executes the query in 100 milliseconds which is 1500 times faster than in Spark. For query 15 with scale factor 20, Spark needs 150 seconds for the evaluation. For scale factor 2 another outlier is query 6 and for scale factor 1 we have an additional one for query 11. For the other scale factors, queries 6 and 11 also produce one of the highest relative speedups compared to Spark. The queries with the smallest relative speedups are 9, 13, and 16. Those three queries are the only queries which are less than 100 times faster than Spark for the scale factors 5 and 10. In case of scale factor 1 and 2, only for query 9 Umbra achieves a relative speedup which is smaller than 100.

In Figure 4.5 we see the relative speedup for two TPC-DS queries. We choose two queries that both contain window functions since these are not covered by the TPC-H queries. In the left Subfigure 4.5a, we see the results of our C++ DataFrame API when used with Postgres and `DataSourceRelation` data frames as starting point. In the right Subfigure 4.5b we use `DatabaseRelation` data frames as starting point which means that the tables already existed in the database previous to the query execution. When the relations are not already stored in the database, the speedup is significantly slower: For all scale factors, executing query 63 takes more time with Postgres than with Spark. In case of query 12, Postgres is able to execute the query faster for scale factors 1 and 2. For the scale factors above 5, the relative speedup or in this case the relative slowdown stays constant. With a growing data size, it seems that Spark is not able to extend its lead. In Subfigure 4.5b we observe



(a) *Postgres (data loaded from CSV files)* **(b)** *Postgres (data stored in the database)*

Figure 4.5: *Relative speedup of selected TPC-DS queries in C++ DataFrame API*

that Postgres is faster than Spark for both queries when the data is already stored in the database. For query 12, Postgres is 30 times faster than Spark for scale factor 1 and still more than 5 times faster for scale factor 20. For query 63 the speedup is smaller. Postgres is 10 times faster than Spark for scale factor 1 and still 3 times faster than Spark for scale factor 20. In contrast to Subfigure 4.5a, the speedup (or slowdown) does not stay constant with a growing data size but gets smaller.

4.3.3 Discussion

Postgres needs a lot of time to load the data into the database before it can start working on the data. Spark skips this step and directly works on the provided CSV files. For most of the queries Postgres is not able to compensate the overhead of loading the data, but for some cases Postgres is still faster. In Table 4.1 we summed up the number of lines of all relations needed per query. In addition, we display the ratio of input rows compared to the overall rows produced during the query evaluation for both Postgres (P) and Umbra (U). We can see that the queries 2, 11, 13, and 16 are those queries where the fewest data has to be read. For scale factor 10, queries 2 and 11 are the only queries where Postgres is faster than Spark. The queries 13 and 16 needed more time in Postgres than in Spark. However, compared to the others they still produce the smallest slowdown. For these queries the number of lines that are read is comparably small when compared to the overall tuples produced during the evaluation. With a ratio of about 20 percent for the queries 13 and 16, it has a smaller effect on the runtime compared to other queries.

We observe that for query 7 Postgres produces the highest slowdown compared to Spark. In Table 4.1 we can see that query 7 is one of the queries where the most data has to be copied into the database before starting the execution. Query 5, 8, and 9 are the only queries where Postgres needs to copy even more data. Nevertheless, Postgres is able to execute these three queries with a smaller slowdown compared to query 7. This shows that the query optimizer is able to compensate the preloading of the data a little. For queries

Query	Relations	Input Rows	Input rows ratio (P)	Input rows ratio (U)
1	1	6,000,000	43.20%	50.56%
2	5	1,010,030	32.26%	32.78%
3	3	7,650,000	63.47%	61.51%
4	2	7,500,000	49.30%	70.85%
5	6	7,660,030	42.13%	52.43%
6	1	6,000,000	98.13%	97.43%
7	5	7,660,025	66.68%	65.29%
8	7	7,860,030	53.07%	54.01%
9	6	8,510,025	40.16%	50.5%
10	4	7,650,025	67.89%	80.04%
11	3	810,025	46.35%	41.42%
12	2	7,500,000	82.39%	74.96%
13	2	1,650,000	23.57%	33.08%
14	2	6,200,000	93.54%	94.56%
15	2	6,010,000	94.04%	96.33%
16	3	1,010,000	22.24%	41.54%
17	2	6,200,000	31.61%	33.35%
18	3	7,650,000	33.33%	25.63%
19	2	6,200,000	96.01%	87.28%
20	5	7,010,025	64.54%	74.45%
21	4	7,510,025	23.15%	36.41%
22	2	1,650,000	34.58%	51.94%

Table 4.1: *Input relation statistics of all TPC-H queries for scale factor 1*

8 and 9, Postgres chooses a different join order than Spark. Spark directly starts with the `lineitem` relation which contains the most rows and immediately joins this table with another relatively large relation. Postgres starts with smaller join partners in both cases. Additionally in the last column of Table 4.1, we can see that for query 7 the ratio of input rows is the highest compared to the queries 5, 8, and 9.

In the second Postgres scenario, the data is already available in the database. The queries 11 and 15 are the ones with the highest relative speedup. When comparing the execution plans of Spark with those of Postgres we see some differences that might cause the speedup: Spark and Postgres sort the result as the last step for query 11, but Spark contains four additional sort operations in its execution plan. Another thing that is noticeable when comparing the execution plans is that Spark splits every aggregate operation into two steps: One for working with partial results and the second to finalize the aggregation. The Spark execution plan therefore contains four aggregate operations, whereas Postgres only contains two.

In case of query 15, Spark scans the biggest file of the whole dataset twice, namely the `lineitem` CSV file, which contains 6 million rows for scale factor 1. In addition, Spark contains six aggregate statements, whereas Postgres only contains three due to the partial and finalized aggregates in Spark.

Our C++ DataFrame API performed much better with the in-memory database Umbra than with Postgres. When we use Umbra together with the `DataSourceRelations` we see a similar behavior in the speedups as for Postgres. As we have already observed for Postgres, we can see a strong correlation between the queries with the highest speedup and the number of rows. The queries 2, 11, and 16 contain the fewest rows, which we can see in Table 4.1. These queries are all outliers of the boxplots in Figure 4.3, which represent the speedup of Umbra over Spark when using `DataSourceRelations`.

As we already observed above, there are some queries which need more time in Umbra than in Spark, especially for a large TPC-H scale factor. The queries 14 and 21 are the slowest in comparison to Spark for scale factor 10. The amount of data that Umbra has to read for query 21 is high compared to the other queries. For query 14, Umbra has to read less data but in Table 4.1 we can see that the ratio of the input rows is almost three times higher than for query 21. The execution plan of query 14 is very short and both Spark and Umbra execute the operations in the same order. This means that in this case only the preprocessing of the data leads to the overhead for Umbra. For query 21 Spark chooses a different join order than Umbra, which might be a better fit for the query.

The best speedups are achieved by Umbra using `DatabaseRelations`. In this case Umbra does not need to preprocess the data anymore, but, in contrast to Spark, Umbra already knows the given data. Spark still has to check the data to be able to generate an optimized execution plan. The lead of Umbra gets smaller with a growing data size but for scale factor 20 the median of Umbras speedup is getting higher again.

Query 15 is the only query where the speedup gets bigger for the growing scale factor. For the scenario which uses Postgres with `DatabaseRelations`, query 15 is also one of

the outliers. However, in contrast to Umbra, Postgres is not able to increase the speedup lead. When comparing the execution plans, we see that Umbra only accesses the `lineitem` table once whereas Spark scans the file twice. The `lineitem` relation is used twice in the execution plan, but both times the same aggregate and filter operation is called on the table. Umbra, therefore, works with a temp view to only execute the identical parts once. In the execution plan of Spark, we can see that the filter and aggregate operations are executed separately on each `lineitem` table scan.

The smallest speedup is achieved for query 9, but since Umbra is still 28 times faster for scale factor 20, we still assume that Umbra has a better execution plan than Spark. In Table 4.1 we can see that for query 9 the most data has to be read, either from the CSV files or from the database relations. This leads to the comparatively similar runtimes of Spark and Umbra. In this query, many relations are joined together. When comparing the execution plans, we observe that Spark and Umbra choose different join orders for the query. The join order of Umbra might be the reason why Umbra is still faster than Spark.

For the TPC-DS queries with window functions, we see a significant difference in the speedups for query 12 and query 63. The reason for that is the amount of data that has to be scanned. In case of query 12, Postgres has to preload 800,000 rows of data for scale factor 1. For query 63 about 3 million rows have to be preloaded. That is why Postgres is only able to outperform Spark for query 12 in the scenario of Subfigure 4.5b. For scale factor 5 and query 12, Postgres has to read more data than for query 63 with scale factor 1. In this case, the speedup of query 12 is smaller than the one of query 63. The execution plans of Postgres and Spark are very similar. Only in case of query 12, Postgres chooses another join order than Spark. In case of query 63, it is only the amount of read data that makes the difference. In the first scenario Spark has an advantage since it directly works on the data. In the second scenario Postgres has an advantage since it already preloaded the data and therefore has better knowledge about it than Spark has.

4.4 How fast can a relational database working with CSV tables answer data frame queries?

4.4.1 Study Design

For the second research question, we run the TPC-H queries in our C++ DataFrame API. This time we use Umbra as database backend since only Umbra supports to directly work on CSV files. Instead of preloading the data or creating tables as part of the query execution, we use the `umbra.csvview()` function. As explained in Chapter 3, we work with a `DataSourceRelation`. When generating the SQL query, we insert calls to the `csvview()` function whenever a query plan of type `DataSourceRelation` occurs. We use the same five scale factors 1, 2, 5, 10, and 20 as for the first research question. To compare the results of the `csvview()` function setup, we reuse the results of Spark we computed for the first research question. The used system in this research question is the same as for the first one.

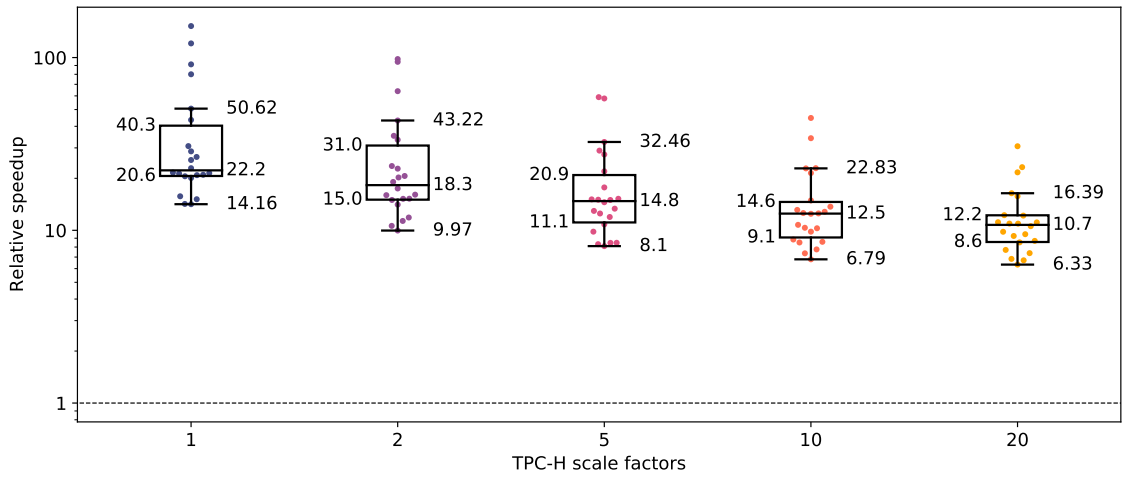


Figure 4.6: *Relative speedup of the C++ DataFrame API with Umbra and umbra.csvview() function over the Spark DataFrame API*

4.4.2 Results

In Figure 4.6 the relative performance of Umbra compared to Spark is shown for the different TPC-H scale factors. We can see that our C++ API with the CSV table scans of Umbra is faster than Spark in all presented scenarios. For scale factor 1 Umbra is at least 14 times faster. For scale factor 20 which means 20 GB of data, Umbra is still able to outperform Spark by at least a factor of 6.3. For scale factors 1 and 2, Umbras speedup is positively skewed and gets smaller for higher factors. Overall, the shapes of the boxplots are similar, but the median speedup gets smaller with a growing data size. When comparing the differences between the median values we see that the speedup loss does not shrink linearly but still gets less with a growing data size.

In all of the five scenarios there exist some outliers where the speedups are higher than the boxplot upper whisker. For all scale factors these include the queries 11 and 16. For scale factor 1, the queries 13 and 22 are also marked as outliers. This does not hold for all of the other scale factors but the speedup is still one of the highest compared to the other queries for all five scale factors. For scale factor 20 query 15 is also an outlier. Similar to the outliers of scale factor 1, query 15 is also one of the queries with the highest speedup for the other scale factors.

4.4.3 Discussion

Umbra used with the `csvview()` function is the most comparable scenario of all scenarios of the first two research questions. In this case, both, the Spark DataFrame API and Umbra, read the data from the CSV table. In Postgres, we have to start with preprocessing the data into database tables, when using the classic `DataSourceRelations`. Therefore, we at first generate database tables and load the data into the tables before starting to

execute the actual query. In Figure 4.3 we can see that the overhead of creating the database relations instead of directly working on the CSV files leads to Umbra being slower than Spark in some cases. In Figure 4.6, as we already observed, all queries are executed faster in Umbra than in Spark; Some of the queries with a significantly higher speedup than others. We explicitly mentioned queries 11 and 16 since these are outliers for all investigated scale factors.

The execution plan of Umbra for query 11 shows that Umbra works with a temp view to avoid scanning the same file twice. In Spark, the `partsupp` relation is scanned twice, which is the biggest relation used for query 11. Since the `partsupp` relation contains 800,000 rows, which is almost have of the total number of rows that have to be read in this query, Spark reads almost twice the amount of data compared to Umbra.

For query 16, Spark and Umbra choose different join orders for the three join partners. In addition, Umbra chooses a hash join for both joins, whereas Spark sorts the data first to perform a sort-merge join. These reasons seem to cause the speedup of Umbra over Spark.

4.5 How time-consuming is the usage of the C++ DataFrame API instead of a SQL interface?

4.5.1 Study Design

The microbenchmark queries for the third research question are executed in our C++ DataFrame API. We only measure the time it takes to call the transformations and to generate the SQL queries. We do not send the generated queries to a relational backend to actually evaluate them. We only want to measure the overhead of a Spark API with a relational database as backend compared to directly accessing the relational database with SQL queries. We vary the size of the queries by repeatedly calling the same function on the same data frame for the operator evaluation of `FILTER`, `AGGREGATE`, `UNION`, and `JOIN`.

Our fifth micro-benchmark evaluates the performance of the expression parsing. We want to measure how fast our program can convert complex expressions into a SQL expression. Repeatedly calling a `FILTER` transformation with the same expression does not make the expressions more complex. To achieve a variable complexity we build one expression and repeatedly combine it with the same expression using the logical `AND` operator. Our program does not check if the two expressions that are combined to one expression with the `AND` operator are the same. Therefore, we use the same expression for all base expressions. In our case the base expression checks if one column contains a certain literal: `"b"_c.contains(lit('abc'))`. The resulting expression for our microbenchmark looks as follows:

```
((("b"_c.contains(lit('abc')) && "b"_c.contains(lit('abc')))
       && "b"_c.contains(lit('abc'))) && (...))
```

As repetition counts we use the logarithmic sequence of base ten from 1 to one million.

In addition to our C++ API, we run the queries in the Spark shell implemented in Scala and compare the execution times. In Spark, no SQL query is generated before it can be

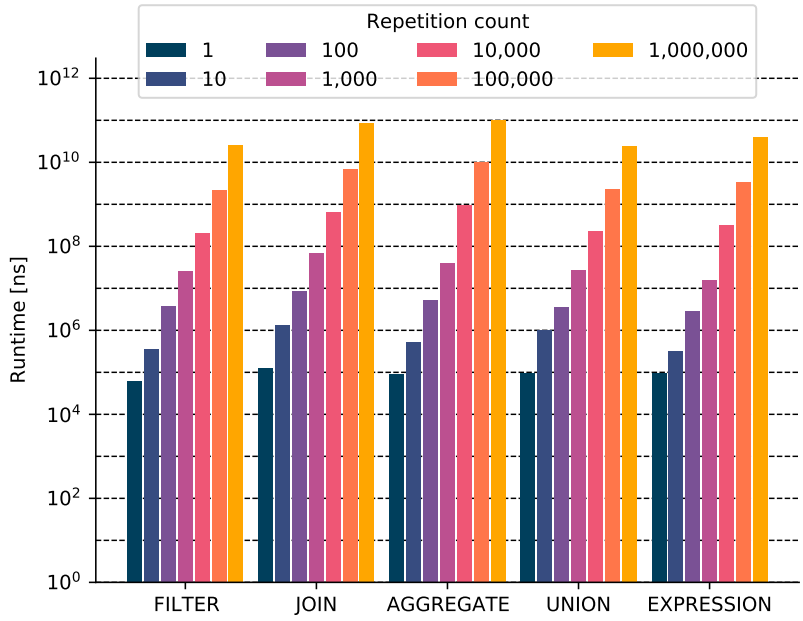


Figure 4.7: *Execution time of generating SQL queries with the C++ DataFrame API*

evaluated. Thus, we add the time our implementation needs to generate the query from the transformations to the overall time it takes to execute the transformations. For this research question we again use the system we already used for the first two questions.

4.5.2 Results

In the third research question, we want to evaluate how much time it takes to generate SQL queries from the DataFrame API transformations. In Figure 4.7 we show the resulting runtimes with a grouped bar chart. Every group represents one of the five microbenchmark scenarios. The first four scenarios deal with the repeated execution of a certain transformation. The different colors represent different repetition counts. We start with one repetition (darkest color) to a repetition count of one million (brightest color). The last scenario represents the evaluation of an expression with an increasing complexity. The different colors in this case represent the number of base expressions of type `CONTAINS` that are connected with the logical `AND` operator.

The y-axis is logarithmically scaled and shows how much time (in nanoseconds) the execution of the different scenarios take. Since we use a base 10 logarithmic scale for the y-axis as well as for the step size of the repetition counts, we can see that the bars are close to linear growth especially between the repetition counts one hundred and one million. This means we have an almost linear runtime. All five scenarios produce very similar runtimes, with the `UNION` and the `FILTER` scenario being the fastest. The execution times of repeatedly executing the `filter()` transformation and the execution time of

an increased expression complexity are also very similar. For some repetition counts, repeatedly calling the `filter()` transformation is faster and for some cases calling the `filter()` transformation only once with a complex expression is faster. The `AGGREGATE` scenario, in which we repeatedly call the `groupBy()` transformation followed by an `agg()` call, is one of the slower benchmarks. The same applies for the `JOIN` benchmark. The `JOIN` scenario gets a better performance compared to the `AGGREGATE` scenario once the repetition count is growing.
After getting a good overview of the runtime relations in Figure 4.7 we can take a closer look at the actual times in Table 4.2. Every column represents a different repetition count and each row is representing the runtimes of a specific microbenchmark. The runtimes for the different scenarios start to differ with a higher repetition count. The `JOIN` and `AGGREGATE` scenarios need more than twice as much time for 100,000 repetitions and 1 million repetitions compared to the other three scenarios `FILTER`, `UNION`, and `EXPRESSION`. The `EXPRESSION` scenario executes faster than the `FILTER` and `UNION` scenarios from repetition count 10 to 1,000. However, it starts to get slower than the other two scenarios above 10,000 repetitions. In the last column, we can see that evaluating the complex expression takes about 15 seconds longer than repeatedly calling the `FILTER` or `UNION` transformation.

In Table 4.3 we list the runtimes of the microbenchmarks in Scala. The scenarios `FILTER`, `JOIN` and `EXPRESSION` all throw a stack overflow error when we are trying to call the transformations 10,000 times and more. The `AGGREGATE` scenario is still able to finish the request for 10,000 repetitions, but it also throws a stack overflow error when we try it with 100,000 repetitions and more. The `UNION` scenario is not able to finish the 10,000 repetitions scenario in under 10 hours, so we stopped it.

In contrast to the C++ version, the Scala runtimes are not growing linearly. There even exist scenarios where more repetitions are faster than fewer repetitions like the filter microbenchmark, where 1 repetitions takes 1 second and 10 repetitions only take 0.5 seconds. Comparing the runtimes of Scala with C++ we see a lot of differences. The `UNION` benchmark is the slowest for Scala while being the fastest for C++. When we compare the runtimes of 1,000 repetitions, C++ is at least 60 times faster than Scala. Computing the `FILTER` and `JOIN` transformations is more than 100 times faster for both. Executing the `UNION` transformation 1,000 times is more than 6,000 times faster. The `AGGREGATE` microbenchmark is the only one that is still executable for 100,000 repetitions in Scala. Our C++ implementation executes the `AGGREGATE` benchmark with 100,000 repetitions more than 300 times faster.

4.5.3 Discussion

In Figure 4.7 we see that the runtimes of the different microbenchmarks are very similar. That is because the SQL queries for the first four transformations are created in a very similar way. The variations between the different types are due to the different complexities of translating a certain transformation type to a SQL query.

The `UNION` operator can be easily translated to a SQL query. We just create a basic query for both child query plans by selecting all columns and then we combine those with

	1	10	100	1,000	10,000	100,000	1,000,000
FILTER	0.06ms	0.36ms	3.72ms	25.18ms	205.56ms	2.2s	25.65s
JOIN	0.12ms	1.35ms	8.64ms	68.34ms	659.97ms	7.08s	86.71s
AGGREGATE	0.09ms	0.53ms	5.28ms	39.56ms	975.75ms	10.07s	104.05s
UNION	0.1ms	1.01ms	3.61ms	26.78ms	230.47ms	2.29s	24.38s
EXPRESSION	0.1ms	0.32ms	2.86ms	16.11ms	326.13ms	3.41s	39.75s

Table 4.2: *Generation time of SQL queries with the C++ DataFrame API*

	1	10	100	1,000	10,000	100,000	1,000,000
FILTER	1.07s	524.2ms	1.77s	3.19s	-	-	-
JOIN	430.28ms	1.16s	1.32s	18.35s	-	-	-
AGGREGATE	1.06s	1.17s	961.4ms	6.84s	343.84s	-	-
UNION	412.34ms	1.18s	2.88s	57.42s	-	-	-
EXPRESSION	473.96ms	507.61ms	1.53s	-	-	-	-

Table 4.3: *Generation time of SQL queries with the Scala DataFrame API*

`UNION ALL`.

The `AGGREGATE` transformation is more complex. We have to create a `GROUP BY` clause for the columns passed to the `groupBy()` function call and the aggregate expression has to be created. In the `SELECT` clause, both the aggregate and `groupBy` expressions have to be listed. This is more time-consuming than just adding a star operator representing all columns from the child query plan like for the `FILTER` operator or the `UNION` operator.

The `JOIN` benchmark is also more complex. After every call to the `join()` transformation, we call the `select()` transformation and only select the columns of the left join partner. We do that to ensure a constant column size even after a large amount of joins.

The `FILTER` transformation microbenchmark and the `EXPRESSION` microbenchmark have very similar runtimes. In both cases, the same amount of base expressions of type `contains()` are transformed to SQL expressions. In the case of multiple `FILTER` transformations, the SQL expressions are combined by stacking multiple subqueries of type `FILTER`. In the case of the `EXPRESSION` scenario, we combine the base expression of type `CONTAINS` with the logical `AND` operator. In both cases, we have some overhead. In case of the `FILTER` scenario we have to generate for each `FILTER` query plan node in the graph a CTE.

In case of the `EXPRESSION` scenario we only generate one CTE that combines multiple base expressions with `AND` operators. The difference between the two options gets remarkable at 10,000 repetitions, which is not a real-world scenario. Therefore, it does not matter for our implementation if we use conjunction or multiple `FILTER` transformations to generate a complex expression.

In their work about Spark [2] Armbrust et al. present the DataFrame API. They state that it evaluates operations lazily. This means that an action has to be called to trigger the

evaluation of the whole data frame query plan. Calling the same transformation for more than 10 times might not be a real-world scenario. However, it is still interesting to see that even when only calling lazy transformations this leads to enough internal computations to result in a stack overflow error when called too often.

For our C++ implementation, we used at first a recursive approach to evaluate the expressions and columns of the different query plans. This led to stack overflow errors in case of too many repetitions. Therefore, we changed our implementation to use iterative approaches in critical places. The transformation of the column objects to SQL expressions during the SQL query generation was for example transformed into an iterative approach. Resolving the unresolved attributes when a transformation is called was initially also done recursively, which we changed as well.

4.6 How well can a relational database optimize queries generated from data frames?

4.6.1 Study Design

For the fourth research question, we compare our generated queries with the original TPC-H queries. We send both the generated and the original queries to the SQL interface of the Postgres and Umbra databases. We compare the execution plans of the generated and original queries and evaluate the impact of the differences by comparing their runtimes. Since both the generated and the original queries produce the same result, they can be optimized in the same way in theory. We check how well the databases can handle the strongly nested generated SQL queries compared to the original SQL queries, which tend to be less nested and more human-readable. Since the creation of the tables is the same for both the generated and the original SQL queries, we preload the tables into the database and only measure the time it takes the databases to run the queries on the existing tables. We used the scale factor 2 for the data, which corresponds to a dataset size of 2 GB. The used system is the same as for the other research question.

4.6.2 Results

In Table 4.4 we present the results of comparing the execution plans of the generated and the original queries for both Postgres and Umbra. Green cells indicate that the query plan optimizer of the database generated the same execution plan. A red-colored field indicates that the execution plans of the original and the generated query differ.

Postgres is able to retrieve the same execution plans from the generated and the original queries for twelve queries. For the other ten queries, Postgres produces different execution plans. Umbra is able to generate the same execution plans for more queries than Postgres. Only five times the execution plan optimizations of Umbra does not result in the same execution plan. Therefore, there exist five queries where Umbra is able to generate the same execution plans, whereas the query optimizer of Postgres produces different results for the generated and the original queries. To see how the execution plans look like, we visualize the execution plans of Umbra for the original and generated TPC-H query 11 in

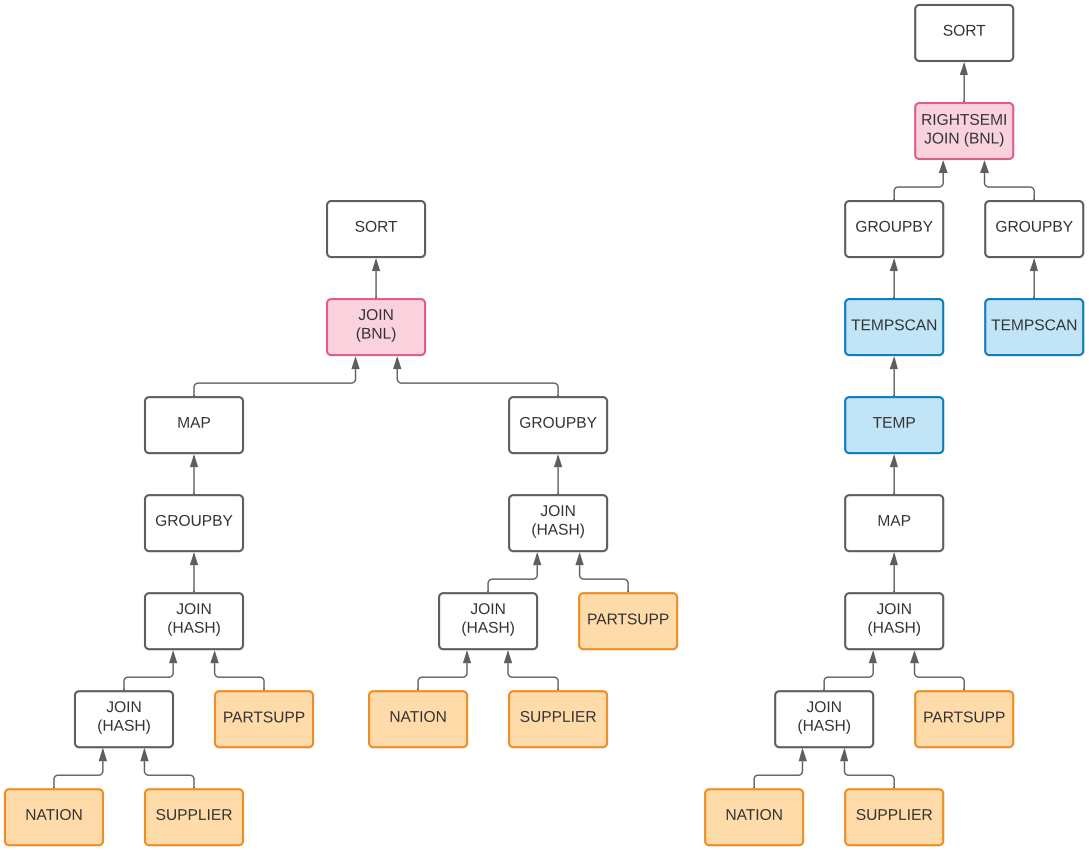


Figure 4.8: *Execution plans for the original TPC-H query 11 (on the left) and the generated TPC-H query 11 (on the right) in Umbra*

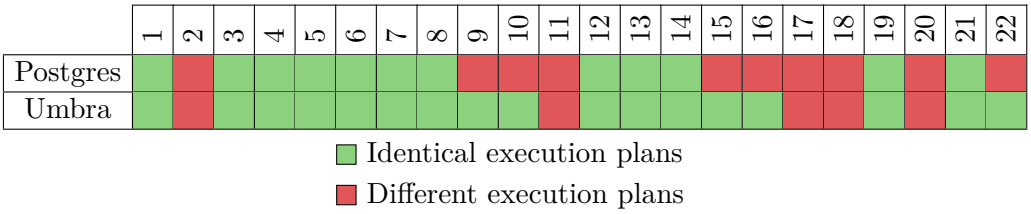


Table 4.4: *Comparison of execution plans of original and generated SQL queries*

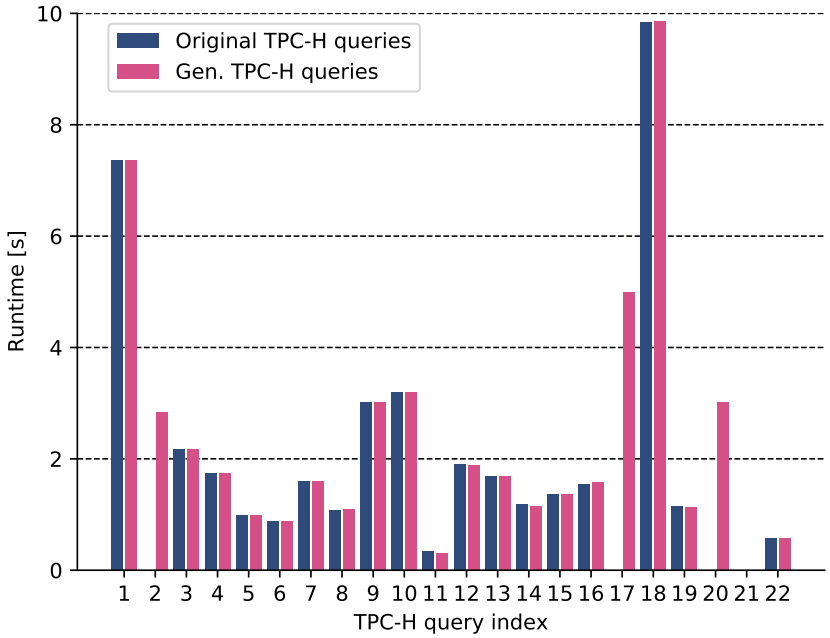


Figure 4.9: *Execution time of original and generated TPC-H SQL queries in Postgres for scale factor 2*

Figure 4.8. On the left we show the original execution plan and on the right the generated execution plan. Orange fields represent database relations and blue boxes contain `temp views` and their usage in `tempscan` operations. The difference between the two execution plans is that for the original query, the Umbra engine scans, joins and groups the three relations `nation`, `supplier` and `partsupp` twice. For the generated query these steps are only done once and then reused twice by the `tempscan` operators. This makes the generated query in this case faster than the original query. In addition, the generated query contains in contrast to the original query a right semi join instead of an inner join.

In Figure 4.9 the execution times needed to run the TPC-H SQL queries in Postgres are shown for the original and the generated queries. The Postgres database is not able to finish the original queries 2, 17, 20, and 21 in less than 10 minutes, so we stopped their execution and left a space in the graph at the corresponding positions. The only generated query which Postgres is not able to finish is query number 21. In contrast to query 21, Postgres is able to finish the queries 2, 17, and 20 when using the generated queries instead of the original ones. Apart from those four special cases, most of the other queries have a very similar runtime when comparing the generated with the corresponding original query execution time. The only remaining query with a percentage deviation of more than 2 percent is query 11. The original query number 11 took 334 milliseconds to execute whereas the generated query needed only 308 milliseconds. This corresponds to a percent

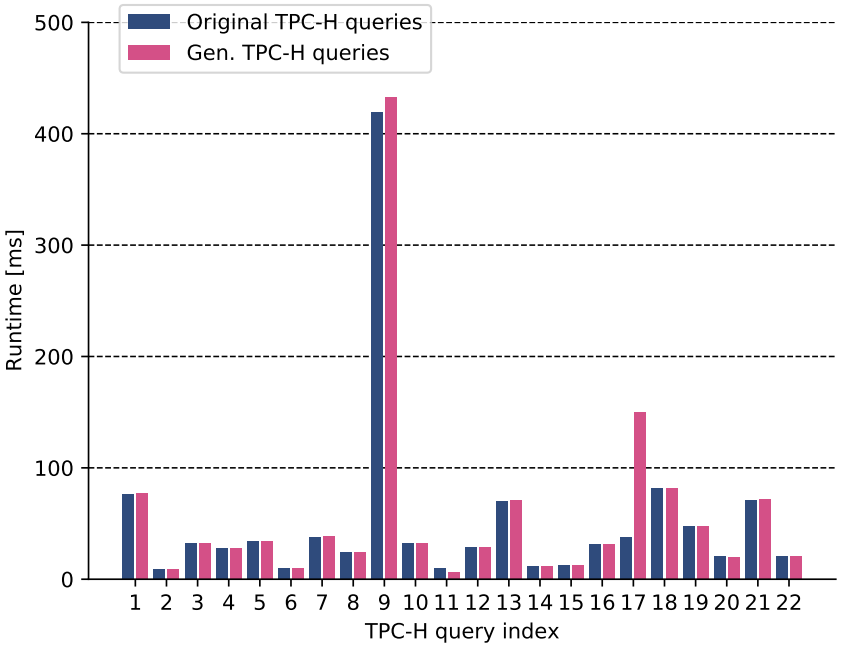


Figure 4.10: *Execution time of original and generated TPC-H SQL queries in Umbra for scale factor 2*

deviation of 8 percent. When we compare these results with the execution plan evaluation in Table 4.4, we can see that the execution plans of the non-terminating original queries differ from the corresponding generated ones which were able to be finished by Postgres. The execution plans of query number 21, where both the original and generated query could not be completed, are the same. Some of the queries have a very similar runtime even though their execution plans differ. That is the case for the queries 9, 10, 15, 16, 18, and 22.

In Figure 4.10 the runtimes for Umbra are shown. Like in Figure 4.9, we show the time it takes to compute the generated and the original SQL queries in the Umbra database for all TPC-H queries. Umbra is able to finish most of the generated queries in the same amount of time as the original queries. 19 of the generated TPC-H queries have a percentage deviation of less than 5 percent. The queries 3, 11, and 17 deviate more significantly from the original queries. Query 3 has a percentage deviation of 15 percent, query number 11 has a percentage deviation of 36 percent, and query 17 takes more than 3 times longer than the original query. The original query 17 only needs 38 milliseconds whereas the generated query takes more than 135 milliseconds which is a difference of nearly 100 milliseconds.

If we compare the runtimes with the execution plans, we can see that different execution plans do not always have an impact on the runtime. This behavior is similar to Postgres. For example, the runtimes of the generated and original queries for query number 5 and

18 are very similar, even though their execution plans differ. The execution time of the generated query has a percentage deviation of less than 1 percent in both cases. Only for query 17 the execution takes significantly longer for the generated query than for the original query.

4.6.3 Discussion

Postgres is not able to compute the result for three of the original queries in a reasonable amount of time. In contrast to the original queries, Postgres is able to finish three of the generated versions in less than 10 minutes. Only query 21 does not terminate for neither the original query nor for the generated query. The other queries, namely 2, 17, and 20, do terminate in the generated case.

When we take a look at the differences in the execution plans, we notice that the original query 2 contains a correlated subquery which we are not able to express in the DataFrame API. To formulate the query in the DataFrame API, we manually decorrelated the query. The query optimizer of Postgres is not able to decorrelate the original query itself. Therefore, the decorrelated generated SQL query terminates in contrast to the original correlated query.

The original query 17 also contains a correlated subquery. In our DataFrame API, we implemented the correlated subquery with a semi-join which results in an `EXISTS` operator in the generated SQL query. Postgres is not able to resolve the original query like we did it manually for the DataFrame API.

Similar to query 17, we also had to manually decorrelate query 20. In the Spark DataFrame API there is no synonym for the SQL `IN` operator. Since query 17 contains this operator twice, we had to rewrite the query. We express the operators in the DataFrame API with semi joins. Like for query 2, the Postgres query optimizer is not able to perform the decorrelation for the original query itself.

The only query which Postgres is not able to execute for neither the original nor the generated form is query 21. The original query contains an `EXISTS` and a `NOT EXISTS` operator, which we transformed into a semi-join and an anti-join. As we described in Chapter 3, we transform semi joins and anti joins into `EXISTS` and `NOT EXISTS` statements. This makes the generated query very similar to the original one. Indeed, in Table 4.4 we can see that the two queries are similar enough to be transformed to the same execution plan by Postgres.

Query 11 was finished for both the generated and the original query. The execution plans differ which is also visible in the execution times in Table 4.9. In the execution plans we notice that the Postgres query optimizer chooses a different join type for the generated query than for the original query. The query 11 contains a `HAVING` clause which is not supported in the Spark DataFrame API. We resolved this clause with a right-semi-join in our implementation which the query optimizer of Postgres turned into a nested loop join. The original query is transformed into a execution plan that only contains hash joins. Using a right-semi-join instead of the `HAVING` clause improves the runtime.

In Table 4.4 there are a few more queries marked as different even though the differences do not have an impact on the execution times. Some of the changes like for the queries 9

or 18 are so small that they are not visible in the execution time. The only difference for query 9 is that Postgres adds an additional sort step to the execution plan of the generated query. In case of the query 18, the execution plan of the generated query contains an additional `MAP` step.

In some cases, the differences in the execution plans are more significant, like a different join type in the execution plans of queries 16 and 22. Nevertheless, the execution times stayed the same. For query 16, that originally contains a `NOT IN` operator, we constructed an anti-join with the DataFrame API. This is still visible in the optimized execution plan of the generated query. The execution plan of the original query contains a `FILTER` expression to deal with the `NOT IN` operator.

Query 22 contains two correlated subqueries. One of the two subqueries is a `NOT EXISTS` operator which we realize with an anti-join in the C++ DataFrame API. Both execution plans contain a "Parallel Hash Anti Join" to realize this correlated subquery. The second subquery is more complex: We had to transform it into a semi-join. The Postgres query optimizer resolves the semi-join into a nested loop with a join filter as one of the last steps. In the original case, the query optimizer is already able to add the correlated subquery as a filter during the parallel sequence scan of one affected table. Nevertheless, both execution plans lead in the end to similar runtimes.

Umbra overall has a better performance in resolving the execution plans than Postgres. However, there are still some differences in the execution plans which are also visible in the runtimes for two queries. Like Postgres, Umbra produces different execution plans for the queries 11 and 17.

The generated query of query number 11 contains a `WITH` clause. As already explained above, we are not able to express a `HAVING` clause with the DataFrame API. Therefore, our version slightly differs from the original one. We create a data frame that joins all tables together so that we can later reuse this data frame twice in two different positions for the main query. The reference count of this data frame is therefore 2. As we explained in Chapter 3, we map these nodes to CTEs, which can be referenced multiple times. The resulting query plan therefore scans the tables only once, whereas in the original case, where they did not use a CTE, the tables are scanned twice. Another difference is the usage of a right semi join in the generated case instead of a join in the original case. This is caused due to expressing the `HAVING` clause in the DataFrame API with a semi join.

In the execution plans of the generated query 17 we can also see that a semi join, in this case a left semi join, is used. As already explained for Postgres, we need to use a semi join to express the query in the DataFrame API. The only difference is that our adaption in the case of Postgres leads to a major runtime improvement. Postgres is not able to finish the original query 17 in less than 10 minutes by Postgres, whereas the generated one only takes about 5 seconds until it is finished. In contrast to Postgres, Umbras achieves a better execution time for the original query than for the generated query. In the original case, Umbra combines a join with an aggregate to a group join in the execution plan. This leads to one instead of two hash tables, hence the aggregate can reuse the hash table of the join operator.

5 Related Work

In her work, Wu [14] compares data frames with database tables. In her opinion data frames have more useful functionalities whereas database tables have a better performance and clarity than data frames. The main difference that she identifies is the difference between multisets used in databases and lists used in data frames. In data frames the user can access a specific row of the data frame by using the relative position as a reference. For database tables, we have to create a subquery in SQL which counts the number of rows with a smaller value for a specific sorting criterion. Since the order of data frames is taken into consideration, some operations are only commutative for database tables. Two database tables with different sorting criteria, are still considered to be equal, whereas a data frame is not equal to the same data frame with a different sorting criterion.
Wu mentions as a drawback for implicit ordering the maintenance costs of preserving the order in a large dataset. For database tables, the user has to add a `ORDER BY` clause to the SQL statement to get a specific order.
Another difference between tables and data frames is the relational algebra which is the base for databases. The user of a database does not have to care about how the data is organized and performs queries on a high-level. Since the user does not influence how the data is organized, endless optimization techniques can be applied in the opinion of the author.

Peterson et al. [10] deal in their work with the performance issues of data frames. They propose different optimization opportunities for the data management of data frames. Their initial intent was to develop a scalable data frame system, called MODIN, by adapting techniques from relational databases. Since data frames are ordered, they state that there is often a strict coupling between the logical and the physical layout required. They present different options for a more lightweight ordering like a separate "order column" if there does not already exist columns which imply the ordering. Instead of keeping the data always sorted internally, the data is only sorted when the program returns a result to the user. In that case, the ordering is applied to the dataset as an `ORDER BY` operator on the "order column". In contrast to a SQL query optimizer, the data frame query optimizer always has to maintain the order in the query plan.
One of the main performance goals mentioned in their work is the support of exploratory data analysis (EDA). In relational databases, EDA is hard to perform since SQL queries cannot be easily executed step-by-step. Users of data frame APIs often debug subexpressions of queries with a trial-and-error strategy, that is why intermediate results are revisited frequently. A data frame query optimizer should consider this to avoid that the same queries are repeatedly executed.
Another consequence of exploratory data analysis they mention is the amount of idle time of the system while the user thinks about what to do next. They present an opportunistic query evaluation technique where the user gets a pointer to a data frame that might be computed in the future. The computation is performed during the thinking time of the

user asynchronously in the background. If the user requests a result, the computation of this result is prioritized by the opportunistic evaluation.
Spark and other data frame implementations like panda mostly return only a prefix or suffix of the result. Therefore, the author suggest to spend the idle time of the system to materialize the prefix and suffix of the data frames.

The following work presents an approach that is similar to ours. The authors present a system that uses a relational database for executing the queries like we do, and they also let the user interact with a more user-friendly API than SQL. The .NET Language-Integrated Query (LINQ) framework presented by Meijer et al. [5] defines general query operators for traversing, filtering, and projecting which can be used as a base for any .NET language to define a special syntax for queries in this language. They present two domain-specific APIs, one for XML and the other one for relational data (DLinq). With DLinq the programmer can manage relational data as objects in the host programming language .NET. The framework translates language-integrated queries into SQL queries and sends them to the database for the execution. The resulting tables are translated back to objects by the framework. The C++ Spark API which we implemented basically does the same thing only the resulting format differs. We work with data frames instead of objects, but like DLinq we transform the data frame queries into SQL queries and send them to the database. Afterwards, we parse the resulting table into the correct format depending on the action the user has called.

Ramnarayan et al. [12] describe in their work how they combine Spark SQL with an in-memory transactional store with scale-out SQL semantics, called GemFire, to one integrated solution which they named SnappyData. Their goal is to provide a unified engine that supports stream analytics as well as OLTP and OLAP. The big data computational engine Apache Spark has in their opinion an appealing programming model to both, application developers and data scientists. The problem is, that Spark has no own storage engine which leads to the need for an external one. In their open-source platform 'SnappyData' they deeply integrated GemFire into the Spark application which improves the performance since the serialization costs and the network traffic can be minimized. By integrating GemFire they were also able to support mutability which is not natively supported by Spark due to the immutable RDD data structures. The in-memory transactional store extends the Spark API with OLTP operations like inserts or updates. Another benefit of the integration is a higher availability due to the replication and fine-grained updates GemFire supports. Mozafari et al. [6] state in their second work about SnappyData that the main idea of their platform is to federate SQL queries between Spark's Catalyst and GemFire's OLTP engine. With an initial query plan, they determine if a query is a low latency query or not. High-throughput analytics are handled by Sparks lineage-based system which is designed for that kind of queries. GemFire deals with the low-latency operations with its consensus-driven replication-based system design.

6 Conclusion

6.1 Summary

Relational databases are still well-known for their great performance on variable size datasets. Nevertheless, the database query language SQL is not very user-friendly. Especially programmers are having a hard time using SQL since they normally use it inside another programming language. Therefore, they build the SQL queries as raw strings without any syntax-highlighting support or type checking support from the IDE. Data frames are an abstraction of tabular data that is closely integrated into different general-purpose programming languages. Developers like that it is possible to generate relational queries step-by-step. Transformations can be invoked on existing data frames and the calls will return new data frames. Thus, the user can check intermediate results and can easily build up complex statements with method-chaining. In SQL, queries can also be built step-by-step but it is not intuitive and very error prone. The user would have to rewrite the query multiple times to finally get the desired query. To check intermediate results, the user has to extract the section of the SQL query that should be calculated, which can be hard due to the nested structure of SQL statements.

To get the best of both worlds, we suggest a mapping strategy to transform data frames to database queries. We implemented a DataFrame API in C++ similar to Spark and whenever an action is invoked on a data frame, we transform it into a SQL query and send it to a database for the evaluation. Our implementation is not restricted to a certain database but can be connected to any database. We support many of the relational transformations of the Spark DataFrame API and are able to express all the TPC-H queries with these. We suggest converting the query plan tree of the data frame into a SQL query by mapping each node of the query plan tree to short SQL queries separately. A parent node can either place the computed subquery itself in its `FROM` clause or the name of the CTE if the child node is referenced by more than one parent node. For ten different query plan types we showed how they can be mapped to a small SQL subquery and how they can be combined to construct even complex queries like the TPC-H queries.

We evaluated our approach with the already mentioned TPC-H queries that we map to a chain of data frame transformation calls. We used two different databases, a disk-based and an in-memory system. In addition, we used different setups to check how the database performs when the data is already stored in the database or when it has to be loaded as part of the query evaluation. We investigated the performance of our approach with four different research questions. These questions address the difference between the execution times of Postgres and Spark as well as the difference between Umbra and Spark. Postgres is not able to beat Spark when the tables are not already loaded into the database. Especially for large datasets, there are only a few queries which need less time for the execution in Postgres than in Spark. If the tables are already available in the database especially Umbra is able to achieve a huge speedup. For all scale factors from 1 up to 20, Umbra outperforms Spark. Even though the relative speedup gets smaller with a growing data

size, Spark is still not able to outperform Umbra with a data size of 20 GB. For some of the queries Umbra is more than 1000 times faster. Utilizing the `csvview` function instead of storing the data in Umbra is the most comparable scenario since Spark does not have an advantage (when the database systems have to load the data into the tables first) or a disadvantage (when the tables are already stored in the database systems) due to fewer precomputations. With the help of the `csvview` function, Umbra is still at least 6 times faster than Spark.

Apart from the higher usability, we also showed that generating a SQL query from a data frame does not take longer than a second even after 10,000 transformations. Some data frame generations can take longer than 100 seconds if certain transformations are invoked one million times, but since the runtime grows linearly it is still acceptable.

In summary, we presented an approach to transparently map data frames to database queries that is able to compete with Spark's DataFrame API on a single machine when it comes to relational data processing.

6.2 Future Work

In the future, we want to further enhance the C++ DataFrame API to make it competitive with the current Spark DataFrame API. We suggest the following areas for improvements:

Distributed database systems. In this master's thesis, we focus on the single node machine for Spark since we only use database management systems that run on a single machine. Spark can also be used for cluster computing. In that case, Spark distributes the data over a cluster and processes queries in parallel. A distributed database system also enables the user to store the data across multiple physical locations. In the future we can investigate how Spark executed on a cluster performs compared to a distributed database system.

User defined functions (UDF). In Spark, the user can work with user-defined functions. He or she defines the function in the same language in which Spark is used as well. In our case, we would specify a Scala function since we work with Spark's DataFrame API in Scala. To use the function as UDF, we call the `udf()` method of Spark and pass our function definition as a parameter. Afterwards, the result of the method call can be applied as a function call to column objects of the DataFrame API. SQL also supports user-defined functions. The user can write these in different languages depending on the used database management system. Postgres for example supports c-language functions, so the user might be able to define a function in C++ and apply it to a DataFrame column object in our C++ DataFrame API. When generating the SQL query, the C++ function would have to be converted into a c-language SQL function for Postgres.

Implicit ordering. The main difference between database tables and data frames is that database tables are treated like multisets whereas in Spark lists are used to store the data. Lists implicitly keep the order of the initial data source. In Postgres for example we could add a column with the `SERIAL` pseudo type which auto increments its value every

time a new row is inserted. All query plan nodes would have to track the current order criteria so that in the final SQL statement the current order criteria can be used in the `ORDER BY` clause. If a query plan node has the type `Sort`, the order expression has to be added at the front of the query plans order criteria. If a column of the order criteria is removed, it has to be removed from the order criteria as well. For the resulting SQL query, we do not have to add a separate CTE for the `Sort` query plan since all other query plans implicitly track the order criteria. Only query plans where the sorting criteria is required, like for a `Limit` or a `Deduplicate` query plan, can insert it into their CTEs.

List of Figures

List of Listings

List of Tables

Bibliography

[1] *Apache spark project*, http://spark.apacke.org.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark", in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.

[3] A. Kemper and A. Eickler, *Datenbanksysteme: Eine Einführung*. De Gruyter Oldenbourg, 2015.

[4] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots", in *IEEE 27th International Conference on Data Engineering*, 2011.

[5] E. Meijer, B. Beckman, and G. Bierman, "Linq: Reconciling object, relations and xml in the. net framework", in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 706–706.

[6] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav, "Snappydata: A unified cluster for streaming, transactions and interactive analytics", in *CIDR*, 2017.

[7] M. F. C. Nazário, E. Guerra, R. Bonifácio, and G. Pinto, "Detecting and reporting object-relational mapping problems: An industrial report", in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, Porto de Galinhas, Recife, Brazil*, 2019.

[8] T. Neumann and M. J. Freitag, "Umbra: A disk-based system with in-memory performance.", in *CIDR*, 2020.

[9] *Pandas*, https://pandas.pydata.org.

[10] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, "Towards scalable dataframe systems", *Proceedings of the VLDB Endowment*, vol. 13, no. 11, pp. 2033–2046, 2020.

[11] *Postgresql*, https://www.postgresql.org/.

[12] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav, "Snappydata: A hybrid transactional analytical store built on spark", in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 2153–2156.

[13] *The r project for statistical computing*, https://www.r-project.org.

[14] Y. Wu, "Is a dataframe just a table?", in *10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, "Spark: Cluster computing with working sets.", *HotCloud*, vol. 10, no. 10-10, 2010.

A TPC-H query 4 (C++ DataFrame API)

```
auto tpch4 = order
    .filter("o_orderdate"_c >= "1993-07-01" && "o_orderdate"_c < "1993-10-01")
    .join(
        lineitem,
        "l_orderkey"_c == "o_orderkey"_c
            && "l_commitdate"_c < "l_receiptdate"_c,
        "semi"
    )
    .groupBy("o_orderpriority")
    .agg(count("*").as("order_count"))
    .orderBy("o_orderpriority");
```

B TPC-H query 4 (Gen. SQL query)

```
SELECT *
FROM (
    SELECT "o_orderpriority", COUNT(*) AS "order_count"
    FROM (
        SELECT *
        FROM (
            SELECT *
            FROM orders
            WHERE (
                (orders."o_orderdate" >= '1993-07-01')
                AND (orders."o_orderdate" < '1993-10-01')
            )
        ) filter3
        WHERE EXISTS (
            SELECT *
            FROM lineitem
            WHERE (
                (lineitem."l_orderkey" = filter3."o_orderkey") AND
                (lineitem."l_commitdate" < lineitem."l_receiptdate")
            )
        )
    ) join4
    GROUP BY "o_orderpriority"
) agg5
ORDER BY agg5."o_orderpriority";
```