TU WIEN Informatics

Integration von Skyline Anfragen in Spark SQL

Effiziente und produktive Integration von Skyline Anfragen in Spark SQL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Lukas Grasmann, BSc
Matrikelnummer 01633007

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Univ.Ass. Dipl.-Ing. Alexander Selzer, BSc

Wien, 31. Jänner 2022

Lukas Grasmann	Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Integrating Skyline Queries into Spark SQL

Efficient and Productive Integration of Skyline Queries into Apache Spark SQL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering und Internet Computing

by

Lukas Grasmann, BSc
Registration Number 01633007

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Univ.Ass. Dipl.-Ing. Alexander Selzer, BSc

Vienna, 31st January, 2022

Lukas Grasmann	Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Lukas Grasmann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. Jänner 2022

Lukas Grasmann

Danksagung

Ich möchte mich an dieser Stelle bei allen Personen bedanken, die mich bei dieser Diplomarbeit unterstützt haben.

Mein Dank gebührt zuallererst Herrn Professor Reinhard Pichler, der sich zur Betreuung dieser Arbeit bereiterklärt hat. Ich möchte ihm für seine Unterstützung, seine Geduld und auch seine konstruktive Kritik über den gesamten Zeitraum von der Themenfindung bis hin zur Fertigstellung danken. Ohne ihn wäre diese Diplomarbeit nicht erfolgreich durchführbar gewesen.

Außerdem danke ich Alexander Selzer, der diese Diplomarbeit mitbetreut und mich mit Ratschlägen unterstützt hat. Ihm möchte ich auch für die tatkräftige Unterstützung bei der Suche nach einem passenden Cluster für die im Rahmen dieser Diplomarbeit durchgeführten Benchmarks danken.

Mein Dank gebührt außerdem dem Team des lbd Clusters der TU Wien für die Bereitstellung des Clusters sowie für die Beratung und den Support bei dessen Verwendung. Bei dieser Gelegenheit möchte ich Dieter Kvasnicka, Giovanna Roda, Liana Akobian und Natalie Kamenik herzlich danken.

Abschließend möchte ich mich auch bei meinen Eltern bedanken, die mich im Laufe des Studiums mit Rat und Tat unterstützt haben.

Acknowledgements

At this point, I would like to thank everybody who has supported me while creating this thesis.

My thanks go first and foremost to Professor Reinhard Pichler, who agreed to advise and oversee my thesis. I would like to thank him for his support, patience, and constructive feedback from finding the topic for this thesis to finishing it. Without him, it would have been impossible to successfully complete this thesis.

Furthermore, I would like to thank Alexander Selzer who gave me advice during this thesis. Additionally, I would like to thank him for his active support in finding an appropriate cluster for benchmarking the software devised in this thesis.

My thanks also go to the team of the lbd cluster of the TU Wien for their advice as well as support regarding the usage of the cluster. I would like to thank Dieter Kvasnicka, Giovanna Roda, Liana Akobian, and Natalie Kamenik most heartily.

Lastly, I would like to thank my parents who supported me during my studies in both advice and action.

Kurzfassung

Skyline Anfragen werden üblicherweise von Datenanalysten und Datenanalystinnen verwendet, um „interessante“ Punkte in Datenbanken oder ähnlichen Datensammlungen zu finden. Sie können in Empfehlungsmaschinen oder für die Visualisierung von „interessanten“ Punkten verwendet werden. Die von Skyline Anfragen verarbeiteten Datenmengen sind häufig groß und in verteilten Datenbanksystemen gespeichert.

Spark ist ein mächtiges, vereinheitlichtes Analyse-System, das gut zur Verarbeitung von großen verteilten Datenmengen geeignet ist. Die Komponente Spark SQL erlaubt es, SQL-ähnliche Datenabfragen als Zeichenkette oder als API-Zugriff zu schreiben. Skyline Anfragen werden in Spark nicht direkt unterstützt. Daher setzen wir uns die **Integration von Skyline Anfragen in Spark** zum Ziel.

Dadurch kombinieren wir die Vorteile von Skyline Anfragen und Spark SQL. Dies erlaubt uns, Skyline Anfragen einfach als eine Erweiterung von SQL zu schreiben und diese mithilfe von Spark auszuführen.

Um dies zu erreichen, zeigen wir, wie Skyline Anfragen integriert werden, sodass sowohl Anfragezeichenketten als auch Zugriffe per DataFrame/DataSet API ausgeführt werden können. Dies beinhaltet auch die PySpark Integration, sodass Skyline Anfragen auch in Python Skripten, die Spark verwenden, benutzt werden können.

Wir implementieren unterschiedliche Algorithmen und Optimierungen, um die Leistungsfähigkeit zu steigern. Dies beinhaltet einen Block-Nested-Loop Algorithmus, einen verteilten Algorithmus auf Basis eines MapReduce-Ansatzes und einen spezialisierten verteilten Algorithmus für Skylines auf Basis von Datenmengen mit fehlenden Werten.

Abschließend führen wir Benchmarks sowohl in einer lokalen Umgebung als auch in einer verteilten Cluster-Umgebung durch, um die Leistungsfähigkeit unserer Implementierung zu erheben. Aus diesen Benchmarks können wir eine gute Leistungsfähigkeit ablesen. Diese ist besser als jene von vergleichbaren Anfragen, die auf „reines“ SQL setzen um dieselben Resultate zu berechnen.

Abstract

Skyline queries are commonly used by data scientist for finding "interesting" data points in databases or other collections of data. They can also be used in recommendation engines or to visualize "interesting" parts of datasets. The datasets processed by skyline queries are often large and stored in a distributed manner.

Spark is a powerful unified analytics engine that is good at handling large, distributed datasets. The Spark SQL component of Spark also allows writing SQL-like queries as strings and using an API to access the data. We, therefore, set the **integration of skyline queries into Spark** as our main goal.

Thereby, we combine the advantages of skyline queries and Spark SQL. This allows us to easily write skyline queries as an extension of SQL and run the query using Spark.

To achieve this, we show how skyline queries can be integrated into both query strings and the DataFrame/DataSet API of Spark. This includes integration into PySpark such that skyline queries can also be executed in Python scripts that use Spark.

We implement different algorithms and optimizations to boost performance. This includes a block-nested-loop algorithm, a distributed algorithm based on common MapReduce approaches to compute skyline queries, and a specialized distributed algorithm which can handle skylines on incomplete datasets.

Lastly, we run benchmarks on both a local and a clustered environment to get a better understanding of the performance of our implementation. From these benchmarks, we can see that the integration gives us good performance figures which are much better than those of equivalent "plain" SQL queries which calculate the same results.

Contents

CHAPTER 1

Introduction

Data scientists as well as software such as recommendation engines, search engines, and booking programs rely on finding "interesting" data points in potentially vast sets of data. This requires a definition of which points are actually "interesting" or important. One idea is to remove all points for which there is at least one other point that is "strictly better". By removing those, only "interesting" data points remain [BKS01]. A data point is "strictly better" if it is "at least as good" in all dimensions and "strictly better" in at least one dimension. This procedure is the main idea of *skyline queries* which define a *dominance* relationship between points in the database and remove those points that are dominated by other points [BKS01].

Skyline queries are a special type of query that is not part of the SQL standard [BKS01]. It is possible to express skyline queries in "plain" SQL but this is impractical for most applications since the queries can become complex. This reduces maintainability and ease of development while potential performance improvements are left on the table. To counter this, there exists a specialized syntax which can be used to express skyline queries in SQL-like languages or in extensions to SQL. The syntax was introduced along with the skyline operator by Börzsönyi, Kossmann, and Stocker [BKS01].

The skyline of a dataset is also a suitable tool for visualization of interesting data points [BKS01]. Skyline queries have additional use cases in different fields such as marketing and economy which make quick and efficient computation of skylines necessary [KT17, HV12]. In the real world, the dataset may also be stored in a distributed manner across multiple database systems. Tools and frameworks for processing distributed data like Apache Spark are therefore beneficial for query processing.

1.1 What are Skyline Queries?

While we have already introduced an intuition to skyline queries by using the notion of "better" data points, we will now proceed to give a more formal definition of skylines. In this section, we will take a look at the most fundamental theoretic foundations as well as a short history overview. Concrete algorithms will be discussed in detail in Chapter 2.

The term *skyline query* was coined as a reference to real-world skylines, i.e., images of cities where the outline of buildings can be seen as a line. An image of the outline of a real-world skyline can also be seen as a combination of points in the dimensions x (horizontal) and y (vertical). For those two dimensions, the y-axis is either maximized or minimized depending on whether the origin of the coordinate system is at the bottom or top of the image respectively. With such skylines, the x-axis is neither minimized nor maximized and must simply have different values instead. Two points with the same value for y axis can only be both part of the skyline if their x values are different. This results in the outline of the buildings as in a regular skyline [BKS01].

If a point is part of the skyline, it can be considered to be potentially "interesting" while all points not included in the skyline are not "interesting". This holds due to the fact that for every point not in the skyline there exists another point that is more "interesting", i.e., part of the skyline [BKS01]. In the words of the skyline paradigm, the data points in the skyline *dominate* the points not in the skyline. To achieve such classification, the skyline query considers multiple dimensions of the data which are usually represented by the columns in the database [BKS01].

Given a dataset P with tuples $p \in P$ and n dimensions, every point in the dataset $p \in P$ can be written as $p = (p_1, p_2, \ldots, p_n)$ where p_i is the value of point p in dimension d_i given that $1 \leq i \leq n$ [KK18]. Given this more formal approach, we can now also give a more formal definition of the *dominance* relationship between two tuples p and q.

Definition 1. *Dominance relationship between tuples [BKS01, KK18]*

Let P be a dataset with n dimensions, let s be a skyline with m relevant *dimensions ($1 \leq m \leq n$), and let $p, q \in P$ be two tuples where $p \neq q$. We assume w.l.o.g. that the first m dimensions (i.e. d_1 through d_m) of the dataset are the dimensions* relevant *for the computation of the skyline since they can be arbitrarily reordered. Then p* dominates *q if and only if for every dimension $d_i \in \{d_1, \ldots, d_m\}$ it holds that p_i is not worse than q_i and there exists at least one dimension $d_j \in \{d_1, \ldots, d_m\}$ such that p_j is strictly better than q_j. The dimensions in $\{m+1, \ldots, n\}$ are not relevant to the skyline computation but* ***are*** *part of the tuples in the skyline since there is no implicit projection as part of the skyline operator. We then denote the dominance relationship by $p \succ q$ if p dominates q.*

This definition of the dominance relationship is dependent on the definition of "better" in the queries. For the purpose of skyline queries, this is usually done by maximizing or minimizing each dimension. Consequently, a value is better if it is higher or lower than the other value respectively [BKS01]. Additionally, a skyline may require a dimension to

be *different* instead of maximizing or minimizing which allows multiple points that are simply different to be part of the skyline [BKS01].

Based on the definition of dominance, we can now also define the skyline itself more formally.

Definition 2. *Skyline [BKS01, KK18, TYA+19]*

Let P be a set of tuples with n dimensions of which m are relevant to the skyline. Then the skyline S is defined as the set of data points $S \subseteq P$ such that for every data point $r \in P \setminus S$ there exists a data point $s \in S$ that dominates r (i.e. $s \succ r$).

We also write $SKY(P)$ instead of S for clarity when referring to the skyline of the set of data points P.

This definition builds on the definition of the dominance relationship as well as the definition of "better" that was already partially discussed. If $p_i = q_i$ for all dimensions $d_i \in \{d_1, \ldots, d_m\}$ for an m-dimensional skyline query, then the tuples p and q may both be part of the (non-*distinct*) skyline. When *distinct* is specified, then **either** p **or** q are retained as part of the skyline. The choice is not specified by this definition [BKS01].

1.2 Why Skyline Queries?

One popular example is already given in the original paper [BKS01]. It encompasses choosing a good hotel for a holiday. Each hotel has different quantifiable properties such as price, distance to the beach, star ratings, and many more which represent the dimensions of each hotel. Finding the optimal hotel is a matter of balancing those properties and finding the best compromise to go with.

The skyline operator helps by eliminating the definitely undesirable options since they are dominated by at least one other option. This is, for example, the case if there exists a hotel that is the same as another hotel in all chosen metrics but does not cost as much. Given the definition of skylines, only hotels remain for which no strictly better hotel exists. This reduces the size of the remaining dataset which can be used for all sorts of recommendation engines as well as search engines. It can also be used to visualize data and reduce the number of data points for further processing.

Integrating skyline queries directly into an SQL-like language makes sense since SQL is a widely known standard. SQL is known by many professions and not limited to software developers only.

A SQL-like syntax was first proposed along with the skyline operator in the original paper [BKS01]. The `SELECT` queries are extended by an optional `SKYLINE OF` followed by the specifications of the skyline dimensions [BKS01]. This syntax was crafted specifically to make it easy to extend existing database systems to support it. While skyline queries can always be expressed in plain SQL, it is beneficial to have a specialized skyline operation

since it prevents errors, cuts down the time needed to formulate the queries, and allows the use of specialized algorithms that are superior in performance.

1.3 Apache Spark

Apache Spark is an open-source unified analytics system that is geared towards distributed computing and distributed databases [Spai]. Spark allows the processing of large amounts of data by using high-level APIs. The parallelization and distribution is handled in a way that works nearly system agnostic [Spai].

There exist multiple APIs for Spark of which the APIs for Scala [Spab] and Java [Spac] are widely used. Additionally, there exist APIs for Python and R [Spaa] as well as specialized libraries such as MLlib [MLl] for machine learning and GraphX [Gra] for graph processing [Spaa].

Spark SQL is a system that allows writing SQL-like queries to easily retrieve results. Originally known as Shark [Sha], it has matured over the years and is now capable of expressing nearly all "regular" SQL queries [Spag]. Due to recent changes in version 3, it is now possible to use a compatibility mode which makes Spark (nearly) fully compatible to the ANSI SQL standard [Spae]. In this mode, all queries will (try to) behave exactly to standard specification regardless of the distributed and parallelized nature of Spark. This includes the supported keywords as well as the behavior with regards to the results of the queries including error handling [Spae]. Spark SQL queries can either be formulated as query strings or via two APIs which are called DataFrame and DataSet.

The Spark SQL system includes the *Catalyst* optimizer which is a powerful rule-based optimizer that speeds up the queries. Optimizations applied are largely agnostic to whether query string, DataFrame API, or DataSet API are used to write the Spark SQL queries [KW17]. Some optimizer rules apply only to certain APIs and are usually used to counter restrictions of the API [KW17].

Spark SQL can be extended by modifying the source code or by using an experimental developer extension API. Since the extension API only offers limited control over the extensions, direct modification of the sources is more promising.

We will take a closer look at Apache Spark, its components, and extending Spark SQL in Chapter 4. For now, we limit ourselves to this coarse overview for purposes of the introduction.

1.4 Goal and Motivation

Skyline queries can be expressed in Spark SQL by transforming them to plain SQL queries as outlined, for example, in Listing 2.4 and Listing 2.5 in Chapter 2. There does not exist any direct way to express skyline queries in Spark since none of the APIs offers this functionality and the Spark SQL syntax does not support skyline queries. This was identified as the problem which we will solve in this thesis.

Given the benefits of both skyline queries and Apache Spark, the goal of this thesis is to integrate skyline queries into Apache Spark and combine said benefits. The functionality is built into Spark by extending it, i.e., directly integrating it into the Spark system. A separate program does not yield the performance necessary since it only uses Spark as a data source and does not take advantage of advanced algorithms or direct parallel computation. If the functionality is implemented in a dedicated program, then users must also acquire that program separately and set it up accordingly which adds additional effort on the user's part.

As additional motivation, we give a short overview over advantages and positive effects of integrating Skyline queries into Spark SQL:

- Integrating skyline queries into Spark SQL allows queries to be formulated in a query-like language. For users already familiar with SQL or a related language, learning to express skyline queries becomes easier.
- Many different types of users from software developers to data scientists use Spark and can benefit from skyline queries.
- Skylines can be implemented to be fast and efficient since they operate directly on the Spark system and do not just use Spark to retrieve the data in its entirety and only subsequently calculate the skyline.
- Skyline queries are regular Spark SQL queries and can build on existing optimizations. This includes the Catalyst optimizer which is a crucial part of the success and efficiency of Spark SQL. More details about the Catalyst optimizer can be found in Chapter 4.
- Skyline queries can be executed on distributed data since Spark is able to handle data from distributed data sources.
- The data for skyline queries can come from many different data sources since many sources are supported by Apache Spark. This includes HDFS, Cassandra, CSV files, and many more.

It follows that extending Spark to integrate skyline queries is the best course of action since it both offers an improved experience to the users and has the potential for much higher performance. This offers a whole new field of possibilities for the users. As Spark is geared towards the use with big datasets and in data science, there is also a big overlap in user bases for Spark and skyline queries.

1.5 Methodology

The methodology we use in this thesis can be split into three phases that can be interleaved as necessary.

A theoretical foundation and understanding of skyline queries is built by literature study. Sources identified during the literature study serve as references in the bibliography for citations. We also identify existing algorithms and optimizations from literature in

this step which will be used to devise algorithms and optimizations suitable for proper integration of skyline queries into Spark.

Based on the literature study, we then devise and implement skyline query algorithms and optimizations based on Apache Spark version 3. This serves as a proof-of-concept for the implementation and can serve as the basis of a more permanent solution of integrating skyline queries into Spark. Devised algorithms and optimizations are also documented in this thesis document.

The performance of the integration is then tested by using benchmarks and tests which are either devised or found as a part of this thesis. Their purpose is to measure performance and prove functionality respectively. The results of the benchmarks are subsequently analyzed.

1.6 Summary of Results

The main result of this thesis is the successful integration of skyline queries into Spark SQL. We now look at a short overview of our results.

- In our literature study, we identify approaches and algorithms that help us integrate skyline queries into Spark. The approaches based on MapReduce are most promising since in Spark, every query is similar to a MapReduce query. The results of the literature study can be found in Chapters 2 and 3.
- Our integration into Apache Spark (Section 5.3) is mainly centered around the easy-to-use component Spark SQL. Given our integration, it is now possible to write skyline queries easily either using an extension of the Spark SQL syntax or the well-known DataFrame/DataSet API. The skyline operator is fully integrated into Spark SQL which includes the parser, analyzer, logical execution plans, Catalyst optimizer, and physical execution plans.
- We integrate skyline queries into PySpark (Section 5.5) such that we can use both query strings and the DataFrame/DataSet API. This enables us to write applications based on skyline queries in Python.
- We provide an implementation of multiple skyline algorithms into Spark (Chapter 5). Each algorithm has a specific strength and use case.
  - The simplest algorithm is a *block-nested-loop* approach which computes the skyline by keeping a window of currently dominating points. New points are added if they are not dominated by any tuple currently in the window. Existing points are removed if they are dominated by new points. This implementation is not distributed and only works on complete datasets. It is especially suitable for calculating skylines of smaller datasets since it consists of only one “step” and does not impose any distribution penalties.
  - Based on the block-nested-loop algorithm, we devise a distributed approach to increase performance for bigger datasets. The distributed approach first

partitions the data into clusters for which the local skylines are calculated independently. They are then used as input for a global skyline computation. This algorithm is best suited for large datasets that are known to be complete.

  - We modify the distributed algorithm such that we are also able to handle incomplete datasets. This vastly expands the number of datasets which can be processed by our integration. We first partition the data based on the `null` values such that tuples for which the same attributes are `null` are assigned to the same partition. For the local skyline, we can then use the "regular" block-nested-loop algorithm described above. To compute the global skyline, we take care of the missing values and avoid problems with cyclic dominance relationships. This variant is best suited for datasets that are either known to be incomplete or where the completeness is unknown. For complete datasets, this algorithm will return the correct results but in most cases perform worse than the other algorithms.
  - Aside from these algorithms, we also devise a way to rewrite skyline queries on incomplete datasets to "plain" SQL (Section 3.2.2). This is an extension over the well-known translation of skyline queries for complete datasets. We use this approach to find "plain" SQL reference queries.

- We provide additional keywords in the skyline query string syntax which can be used to force Spark to use a particular skyline algorithm (Section 5.1.1). Internally, we automatically detect an appropriate algorithm based on whether the skyline dimensions are `nullable` if no keyword is used to override the selection. The overrides are particularly useful for datasets that are known to be complete despite the columns being `nullable` to Spark. In this case, we can then force Spark to still use the complete algorithm.
- Spark's famed Catalyst optimizer provides an opportunity for further improving our integration. We implement multiple rule-based optimizations which make some skyline queries faster (Section 5.4 and Section 5.8). This includes removing redundant or empty skyline dimensions, pushing skylines through joins, and optimizing single-dimensional skylines by using an alternative execution plan.
- We ensure that our integration works correctly by devising and executing both unit tests (Section 6.1) and system tests (Section 6.2). Unit tests are used to check whether individual parts of the integration (e.g., as the modified parser) work correctly while system tests are used to check the functionality of the full implementation stack. In the system tests, we process entire datasets and subsequently compare the results to those of reference queries written in "plain" SQL.
- The performance of our integration is measured through a series of benchmarks on both real-world and synthetic datasets (Section 6.3). We find that all algorithms outperform the reference queries written in "plain" SQL in almost all cases. Given a complete dataset, the distributed complete algorithm outperforms both the block-nested-loop and the distributed incomplete algorithm. The only exceptions are very small datasets for which the block-nested-loop algorithm is sometimes faster.

The sources of our integration, which is a modified version of Apache Spark, are available as open source on GitHub at https://github.com/Lukas-Grasmann/Spark_3.1.2_Skyline/. Other sources used for for setup, testing, and benchmarks can also be found on GitHub at https://github.com/Lukas-Grasmann/Spark_Skyline_Utilities/.

Our integration already offers a system capable of productive usage of skyline queries. It also provides a good starting point for future work beyond what was possible within the scope of this thesis.

1.7 Organization of the Thesis

The rest of the thesis is structured as follows: In Chapter 2 we discuss the theoretical foundations of skyline queries that go beyond what was already discussed in the introduction. In Chapter 3 we discuss more advanced and optimized algorithms which are more useful for large sets of data than the basic algorithms discussed in Chapter 2. After that, we complete the foundations needed for understanding the implementation part of this thesis by introducing Apache Spark in more detail in Chapter 4. Based on the foundations laid out in the previous chapters, we then describe the implementation of skyline queries in Apache Spark in Chapter 5. The evaluation of the performance of the implementation will be the topic of Chapter 6 in which we discuss and compare the results we get from the implementation. Lastly, we will draw a conclusion and look at potential future work in Chapter 7 to round off the thesis.

CHAPTER 2

Skyline Queries

In this chapter, we will take a look at the history and basic algorithms of skyline queries as well as some more advanced theoretical foundations that are needed for more advanced algorithms. We also discuss some of the basic algorithms in this chapter. The algorithms discussed are either historically important or provide a foundation for more advanced algorithms which will be introduced in Chapter 3.

2.1 History of Skyline Queries

We will now take a quick look at *top-k* queries which are older than skyline queries but are based on a similar idea. After that, we will take a look at the skyline operator and the early research on it.

2.1.1 Top-k Queries

Some of the basic ideas of skyline queries are already present in top-k queries. Top-k queries are also a tool to retrieve the most relevant or first data points from a potentially large (ordered) result set [CK97]. As such, top-k can also be used to efficiently obtain the most "relevant" data points similar to the goal of skyline queries.

Retrieving the top k results is potentially inefficient. Computing the entire result set and then taking the k best answers is usually not performant enough for big datasets [CK97]. Optimizations are therefore necessary to achieve sufficiently good performance for this type of query [CK97].

The use cases for top-k queries are exceedingly similar to the ones of skyline queries as they can be used in search and recommendation engines as well as in various areas of data science. Applications in recommendations and searches are intuitive since, e.g., travel websites often return only a limited number of "best" results for each user query sorted by some criteria [CK97].

Naive top-k queries can be achieved by combining ordering according to the scoring function and limiting the number of results to k in SQL queries. Such queries usually benefit from optimizations performed by the database engine that transforms the naive order and limit to more sensible execution plans that do not require the entire result set to be computed before applying the limit. Alternatively, the `STOP AFTER` k command may be used in database systems that support it [CK97]. Which method is best for a top-k query heavily depends on which database system is used.

Similar to skyline queries, top-k queries also get more complex when more dimensions of data are involved. This is especially true if indexes can be used for lower-dimensional top-k queries while they are not available for queries with a large number of dimensions.

2.1.2 The Skyline Operator

The skyline operator is based on a similar idea to top-k queries and specifically tailored for the use in relational databases [BKS01]. Due to this, a SQL-like syntax for skyline queries is already given in [BKS01] and serves as the basis for further development. Along with the theoretical foundations and the syntax, the original paper also introduces some basic algorithms which include:

- Block-nested-loop algorithms
- Divide-and-conquer algorithms
- Index-based algorithms (B-tree, R-tree)

The majority of the research in the field of skyline queries which extends beyond that basic foundation is mainly geared towards big datasets and distributed processing of the skyline queries (e.g., [KK18, CLX+08, LZLL07, TYA+19]). There are also some variants of the skyline queries which are slightly different like reverse skyline queries [DS07]. For some special cases such as incomplete data or streaming environments there also exist algorithms which can start to derive a skyline even if not all data is available [GAT19].

There have been various surveys of the current state of the art of skyline queries. Most of them focus on the state of the art for skyline queries in distributed systems. Distinctions are often made between peer-to-peer and centralized systems and according to the degree of "sharedness" that is prevalent in the respective systems. The categories of "sharedness" include shared-nothing, shared-everything and hybrid systems that are a combination of the two other categories with various aspects (e.g., shared memory, shared computation unit, ...) that can be shared to different degrees in the system [KT17, HV12].

2.2 Skyline Queries in SQL and Related Languages

As already discussed in Chapter 1, there exists a specialized SQL-like syntax for skyline queries [BKS01]. It makes it possible to quickly and effectively formulate skyline queries if they are supported by database system. The syntax of such skyline queries can be found in Listing 2.1 and was taken from [BKS01].

Listing 2.1: Syntax of skyline queries in SQL [BKS01]

```
1 SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...
2 SKYLINE OF [DISTINCT] d_1 [MIN | MAX | DIFF], ..., d_m [MIN | MAX | DIFF]
3 ORDER BY ...
```

In line 2 of Listing 2.1, d_1 through d_m denote the dimensions of the skyline queries, i.e., the columns from the database that are relevant to the query. We note that the data may have additional dimensions which are not relevant to both skyline processing and syntax. In the hotel selection example from above, the skyline dimensions may include the price, distance to the beach, or the star rating of the hotel. Other dimensions such as the name and address of the hotel can be part of the dataset but are not relevant to the skyline and therefore not part of the skyline dimensions. The keywords `MIN`, `MAX`, and `DIFF` define which version of "better" is used. `MIN` and `MAX` stand for minimization and maximization of the dimension respectively while `DIFF` indicates that no dominance between two tuples exists if the values in the `DIFF` dimension are different [BKS01].

The `DISTINCT` keyword specifies the case of multiple data points which are equivalent for every dimension relevant to the skyline computation is handled. If distinct is **not** specified, then all of them are be retained. Otherwise, only one is retained by arbitrary choice [BKS01].

Recalling the example of a real image of a skyline from Chapter 1, we note again that the x and y axes can be seen as two dimensions. This example is particularly useful for understanding the `DIFF` keyword. Two points in outlines of different buildings with different x coordinates may both be part of the skyline and the a `DIFF` keyword is therefore appropriate for this dimension [BKS01]. The use cases for `MIN` (e.g., for prices or distances to the beach) and `MAX` (e.g., for star ratings of the hotels or square meters of the hotel rooms) are more straightforward since they usually match with the intuition of the users.

Given the syntax of skyline queries in Listing 2.1, we can now give an example for a skyline query that computes the skyline of the image in Listing 2.2. Here, we assume that the points of the buildings are in a table `buildings` and the origin of the coordinate system is at the bottom of the image and the y-axis must therefore be **maximized**. If the origin was at the top left corner of the image, we would have to minimize the y-axis instead.

Listing 2.2: Skyline query example for building skylines in an image

```
1 SELECT * FROM buildings SKYLINE OF x DIFF y MAX;
```

Details of skyline algorithms will be discussed later in this chapter and in Chapter 3 which is dedicated to advanced skyline algorithms.

2.3 Formal Skyline Definitions

In this section, we give more formal definitions compared to the ones given in Chapter 1. We formally define skyline queries as well as some basic properties of the skyline queries.

2.3.1 Formal Definition of Skylines

In Chapter 1, we have relied on the abstract notion of "better" to define skyline queries. On one hand, this is a good way to intuitively describe the dominance relationship and the skyline itself. On the other hand, it is disconnected from the syntax of the skyline query formulation and therefore insufficient for formal reasoning and implementation purposes. The skylines and tuples need to be split into multiple parts to give a formal definition of "better". We then give a definition of the skyline preference formula and dominance relationship based on [Ede09].

Definition 3. *Skyline preference formula and dominance relationship [Ede09]*

Given two n-ary tuples r, s with a set of dimensions D, we define four disjoint (potentially empty) subsets of dimensions $D_{\max}$, $D_{\min}$, D_{diff}, D_{extra} where $D_{\max}$, $D_{\min}$, $D_{\text{diff}}, D_{\text{extra}} \subseteq D$. For every dimension $d_i \in D$ where $1 \leq i \leq n$, it holds that $d_i \in D_{\max}$ if d_i is maximized, $d_i \in D_{\min}$ if d_i is minimized, $d_i \in D_{\text{diff}}$ if d_i is `DIFF`*, and $d_i \in D_{\text{extra}}$ if d_i does not influence the computation of the skyline. We use r_i and s_i to denote the value of r and s in dimension d_i respectively. The preference formula $C(r, s)$ between r and s is defined as follows:*

$$C(r,s) := \Big(\bigwedge_{d_i \in D_{\min}} r_i \leq s_i\Big) \wedge \Big(\bigwedge_{d_i \in D_{\max}} r_i \geq s_i\Big) \wedge \Big(\bigwedge_{d_i \in D_{\text{diff}}} r_i = s_i\Big) \wedge \Big(\Big(\bigvee_{d_i \in D_{\min}} r_i < s_i\Big)\Big) \vee \Big(\bigvee_{d_i \in D_{\max}} r_i > s_i\Big)\Big)$$

The dominance relationship $r \succ s$ (r dominates s) is then defined by the preference formula as follows:

$$r \succ s \qquad \texttt{iff} \qquad C(r,s)$$

Intuitively, this restates the definition from Chapter 1. As seen in the first line of the preference formula, a tuple r can only dominate a tuple s if it is at least as good in all `MIN/MAX` dimensions and it is the same in all dimensions with a `DIFF`. Additionally, it must also be strictly better in at least one `MIN/MAX` dimension. This condition is found in the second line.

Given the definition of dominance relationship, we now define the relationship between tuples without a dominance relationship between them.

Definition 4. *Incomparable relationship [Ede09, TYA+19]*

Given two tuples r and s, the tuples are incomparable *if and only if $r \not\succ s$* ***and*** *$s \not\succ r$.*

We note that two tuples which are equal in all dimensions relevant to the skyline are also incomparable according to this definition. This case often needs to be handled separately regardless, e.g., for the distinct skylines.

To ease the notation for future definitions and algorithms, we assume from this point onward that the dimensions of tuples are ordered according to the specified `MIN/MAX/DIFF` in the query. The tuples r and s are written as follows [Ede09]:

$$r = (\overbrace{r_1, \ldots, r_j}^{\texttt{MIN}}, \overbrace{r_{j+1}, \ldots, r_k}^{\texttt{MAX}}, \overbrace{r_{k+1}, \ldots, r_m}^{\texttt{DIFF}}, \overbrace{r_{m+1}, \ldots, r_n}^{\text{extra}})$$
$$s = (\underbrace{s_1, \ldots, s_j}_{\texttt{MIN}}, \underbrace{s_{j+1}, \ldots, s_k}_{\texttt{MAX}}, \underbrace{s_{k+1}, \ldots, s_m}_{\texttt{DIFF}}, \underbrace{s_{m+1}, \ldots, s_n}_{\text{extra}})$$

This notation helps with simplifying future formal definitions and algorithms but does not immediately capture that every dimension set may be empty. It can be used to express all skyline queries since the skyline dimensions can be reordered arbitrarily without changing the results.

Given the more formal definition of dominance, we can now give a more formal definition for the skyline operator.

Definition 5. *Skyline operator [Ede09]*

Let $\mathcal{R}$ be a relational schema and let C be the skyline preference for the schema. Then for every instance R of the relational schema $\mathcal{R}$, the skyline operator with regards to preference C is denoted by $\mathrm{SKY}_C(R)$*:*

$$\mathrm{SKY}_C(R) := \{r \in R \mid \nexists s \in R \ : \ s \succ r\}$$

2.3.2 Properties of Skylines

One of the most important properties of the skyline query is that for each monotone scoring function over the same dimensions it holds that if $p \in \mathcal{M}$ maximizes a monotone scoring function $\mathcal{M} \to \mathbb{M}$ then p must also be a part of the skyline [BKS01].

Skylines are also transitive. We can split the data set P into m subsets $P_{\mathrm{SUB}_i} \subseteq P$ where $1 \leq i \leq m$ and $P_{\mathrm{SUB}_i} \cap P_{\mathrm{SUB}_j} = \emptyset$ if $i \neq j$. Then compute the skylines $\mathrm{SKY}(P_{\mathrm{SUB}_i})$ for every subset P_{SUB_i}. When we compute the "skyline of skylines" $\mathrm{SKY}(\bigcup_{1 \leq i \leq m} P_{\mathrm{SUB}_i})$ then the resulting skyline is equivalent to the skyline of the full original dataset P, i.e., $\mathrm{SKY}(\bigcup_{1 \leq i \leq m} \mathrm{SKY}(P_{\mathrm{SUB}_i})) = \mathrm{SKY}(P)$.

Proof. For this proof, we look at the two cases where we assume that data points exist such that the results of the skyline computation are **not** equivalent. Then we prove by contradiction that such cases cannot exist.

Assume that there is a point $p \in P$ for which $p \in \text{SKY}(\bigcup_{1 \leq i \leq m} \text{SKY}(P_{\text{SUB}_i}))$ and $p \notin \text{SKY}(P)$ hold. Given the definition of $\text{SKY}(P)$, there must exist a data point $q \in P$ in the skyline for which $q \succ p$ holds. Since q is part of the skyline, it follows that $\nexists r \in P \ : \ r \neq q \wedge r \succ q$. Since $\nexists r \in P \ : \ r \succ q$, the point q must also be part of one of the local skylines and, in turn, it must also be part of the "skyline of skylines". Since q is part of the "skyline of skylines" and $q \succ p$ it follows by definition of the skyline via the domination property that $p \notin \text{SKY}(\bigcup_{1 \leq i \leq m} \text{SKY}(P_{\text{SUB}_i}))$ which contradicts our assumption.

Next, we assume that there exist a point $p \in P$ for which $p \notin \text{SKY}(\bigcup_{1 \leq i \leq m} \text{SKY}(P_{\text{SUB}_i}))$ and $p \in \text{SKY}(P)$ hold. Given that $p \in \text{SKY}(P)$ it follows that $\nexists q \in P \ : \ p \neq q \wedge q \succ p$ or $\forall q \in P \ : \ p \neq q \rightarrow q \not\succ p$ by definition of the skyline query. Since there exists no dominating tuple according to this observation, p is part of at least one local skyline. It also follows that $p \in \text{SKY}(\bigcup_{1 \leq i \leq m} \text{SKY}(P_{\text{SUB}_i}))$. This contradicts our assumptions. □

The transitivity of the skyline operator is useful for many algorithms since it allows the reduction of the size of the dataset per skyline calculation by calculating the skyline of a "local" dataset first. This is mostly used to devise algorithms where multiple local skylines are computed in parallel before the global skyline is computed using the results of the local skylines. Such approaches are, for example, found in the following papers: [KK18, CLX+08, TYA+19].

2.4 Basic Skyline Algorithms

In this section, we introduce the most basic algorithms for skylines which were introduced early in the history of skyline queries. While simple and relatively easy to implement, they may not offer the performance necessary to be useful in distributed environments or when handling big data. We also note that not every algorithm is suitable for all database systems or processing frameworks since they may go against the fundamental architecture of said systems or frameworks. This, for example, holds for systems that do not support indexes like most MapReduce architectures.

2.4.1 Direct Translation into Nested SQL Queries

Skyline queries can be transformed into nested SQL queries that are then executed like regular SQL queries on the database. For any skyline query, there exists a corresponding standard SQL query that gives the same results as the skyline query [BKS01]. This approach can serve as a simple basis for implementations but is not viable in production since it shows poor performance.

There are three main reasons for the lack of performance [BKS01]:

- The query corresponds to a naively implemented nested-loop and cannot be "unnested".
- If the skyline includes a `JOIN` or `GROUP BY` then those must be executed in the outer query **and** the subquery.
- Skyline queries can be combined with other operations such that the computational costs for the skyline itself are greatly reduced. This is not easily possible for direct translation.

The query translation is mainly based on `WHERE NOT EXISTS` since this can be used to check for each data point **whether no dominating data point exists**. Therefore, we use the SQL `SELECT` statement to get the set on which the skyline query is computed and then remove dominated data points in the `WHERE` part of the query [BKS01]. We note that performance optimizations may be performed by the database engine itself such that the query is not necessarily executed as written down. The actual execution plan may not match this syntax and intuition but may have vastly superior performance.

Before giving a general approach to translation, we give an example for a skyline query that was also introduced in [BKS01]. It selects hotels in Nassau, Bahamas, where both distance to the beach (`h.distance`) and the price (`h.price`) of the hotel are dimensions relevant to the skyline query. Both dimensions are minimized since both shorter distances to the beach and lower prices are desirable. The SQL query taken from [BKS01] can be found in Listing 2.3 in a slightly modified form.

Listing 2.3: Skyline query as nested query in "pure" SQL [BKS01]

```
1  SELECT *
2  FROM hotel h
3  WHERE h.city = 'Nassau' AND NOT EXISTS (
4      SELECT *
5      FROM hotel h1
6      WHERE h1.city = 'NASSAU' AND
7          h1.distance <= h.distance AND
8          h1.price <= h.price AND
9          (h1.distance < h.distance OR h1.price < h.price)
10 );
```

Given the simplified notation $a = (\overbrace{a_1, \ldots, a_j}^{\texttt{MIN}}, \overbrace{a_{j+1}, \ldots, a_k}^{\texttt{MAX}}, \overbrace{a_{k+1}, \ldots, a_m}^{\texttt{DIFF}}, \overbrace{a_{m+1}, \ldots, a_n}^{\texttt{extra}})$ for a tuple, we can now devise a general schema for translating a skyline query to plain SQL. We first give the corresponding query in skyline syntax in Listing 2.4.

Listing 2.4: Original skyline query [BKS01, Ede09]

```
1 SELECT column_list FROM relation AS r WHERE condition(s)
2 SKYLINE OF a_1 MIN, ..., a_j MIN, a_{j+1} MAX, ..., a_k MAX, a_{k+1} DIFF, ...,
    a_m DIFF;
```

Next, we transform the (simplified and generalized) query in skyline syntax from Listing 2.4 to a (nested) query in plain SQL. The corresponding query can be found in Listing 2.5. In this query, we add an additional condition to the existing `WHERE` conditions of the query (line 1). We only select a tuple as part of the results if there does not exist (`WHERE ... NOT EXISTS`) a dominating tuple (line 2 - 9). To find the dominating query, we use the original query (sans skyline) again (line 2) where we select only tuples which are at least as good for all minimized and maximized dimensions (line 3 and 4) and equal in all `DIFF` dimensions (line 5). Finally, we also ensure that the dominating tuples are better in at least one dimensions for minimization and maximization (line 7 and 8).

Listing 2.5: Translated skyline query in plain SQL [BKS01, Ede09]

```
1  SELECT column_list FROM rel AS o WHERE condition(s) AND NOT
     EXISTS(
2      SELECT * FROM rel AS i WHERE condition(s)
3          AND i.a_1 ≤ o.a_1 AND ... AND i.a_j ≤ o.a_j
4          AND i.a_{j+1} ≥ o.a_{j+1} AND ... AND i.a_k ≥ o.a_k
5          AND i.a_{k+1} = o.a_{k+1} AND ... AND i.a_m = o.a_m
6          AND (
7              i.a_1 < o.a_1 OR ... OR i.a_j < o.a_j
8              OR i.a_{j+1} > o.a_{j+1} OR ... OR i.a_k > o.a_k
9          )
10 );
```

2.4.2 One-Dimensional and Two-Dimensional Skylines

Given the number of dimensions of skyline queries, some special cases can be derived for which (more) efficient algorithms exist. This is opposed to most algorithms described in this thesis which focus on skyline queries with an arbitrary number of dimensions. Performance in queries with an arbitrary number of dimensions may vary greatly especially for high-dimensional skylines.

The first and most notable special case is the one-dimensional skyline query. Since there is only one dimension, it suffices to sort the data and then take the top results (or one of the top results if the query is to be distinct). Due to the transitivity of the skyline operator, this approach may also be split into multiple consecutive steps where for each local skyline only the top entries are retrieved transitively.

The second special case involves skyline queries with two dimensions relevant to the computation of the skyline. Two-dimensional skyline queries can be solved by topologically sorting the data by their two dimensions relevant to the skyline. For every data point, it is only necessary to compare it to its predecessor or, more precisely, to the last point that is part of the skyline [BKS01].

This approach does not work for skyline queries with more than two dimensions. There is an algorithm for solving skyline queries with three dimensions [KLP75, BKS01] but none of the specialized algorithms generalize to an arbitrary number of dimensions [BKS01].

2.4.3 Block-Nested-Loop

The block-nested-loop skyline algorithm is closely related to the common block-nested-loop algorithm that is used in database systems. While executing the nested loop, the results of the skyline can be computed by keeping windows of tuples that dominate other tuples.

2.4.3.1 Basic Block-Nested-Loop Algorithm

For each new tuple t there are the following possibilities [BKS01]:

- Tuple t is dominated by a tuple in the window. Then t is eliminated and no further comparisons are necessary.
- Tuple t dominates one or more tuples in the window. In this case, the dominated tuples are eliminated. Then t is inserted into the window.
- Tuple t is incomparable to the tuples in the window. Then t is written into the window if there is enough room or written to a temporary storage or to disk. These tuples will be processed in further rounds of the algorithm. If the window is empty, then the tuple will always be inserted into the window (i.e., at the start of the algorithm).

After each iteration, the tuples in the window that were compared to all tuples in the temporary file may be put out since they are neither dominated nor do they dominate other tuples. We keep track of the tuples by a simple timestamp-based measure [BKS01].

We now give the best and worst case of the algorithm depending on the number of input tuples n. The complexity of this algorithm is $\mathcal{O}(n)$ in the best case when the entire skyline fits in the window and the algorithm terminates in two iterations [BKS01]. In the worst case, we need to write to the disk every time and add another iteration. The complexity, in this case, is $\mathcal{O}(n^2)$ [BKS01]. It follows that this algorithm works particularly well if the entire skyline fits in the window [BKS01].

2.4.3.2 Variants and Improvements to Block-Nested-Loop Algorithm

One obvious improvement of the block-nested-loop algorithm described above is the optimization of the list that represents the window [BKS01]. This list can be made

self-organizing by moving tuples that are found to dominate other tuples to the top of the list. If tuple t dominates a tuple in a comparison then it is always moved to the top of the list such that the next tuple is compared to t first and then the rest of the list. This variant is especially useful for skewed data with *killer tuples* that dominate many other *neutral tuples* [BKS01].

Another improvement is to use the idea of a self-organizing list but use a Least-Recently-Used (LRU) replacement policy to replace data in the window. This comes at the cost of additional CPU resources to determine the replacement victim. A tuple replaced from the window is written to the end of the temporary file and needs to be compared to the other tuples placed before it [BKS01]. As such, tuples may be compared twice. There are tradeoffs to be considered when implementing such a policy in the self-organizing list [BKS01].

2.4.4 Divide-And-Conquer

Divide-and-conquer approaches differ greatly in details depending on the implementation and basic principles. They have been proposed mainly to be more efficient in a restricted database context where resources like memory are scarce. At the time of the writing of the main skyline operator paper [BKS01], it also was the most efficient algorithm with regards to the worst-case complexity which is in the order of $\mathcal{O}(n \cdot (\log n)^{d-2}) + \mathcal{O}(n \cdot \log n)$ where n is the number of input tuples and d is the number of dimensions relevant to the skyline computation [BKS01].

2.4.4.1 Basic Divide-And-Conquer Algorithm

The basic algorithm works as follows [BKS01]:

- Compute median m_p of the input for some dimension d_p. Divide the input into partitions P_1 whose tuples' values for d_p is better than m_p and P_2 which contains all other tuples.
- Compute skylines S_1 of P_1 and S_2 of P_2 by recursively applying the algorithm. Recursion can stop once the skyline is trivial since the partition contains only few (or one) tuples.
- Compute the overall skyline by *merging* the results of S_1 and S_2 via the elimination of the tuples in S_2 that are dominated by a tuple in S_1. The tuples in S_1 cannot be dominated by the tuples in S_2 since S_1 is better in dimension d_p than every tuple of S_2.

For the merging steps, there exist multiple algorithms. Usually the merging function is applied recursively and terminates once all dimensions have been considered or one of the partitions is empty or contains only one tuple [BKS01].

2.4.4.2 M-Way Partitioning

The general divide-and-conquer approach behaves badly if the input does not fit in the memory [BKS01]. It makes sense to partition the data into m partitions such that every partition does fit into the memory. To achieve this, we compute (α-)quantiles instead of the median as the boundaries for the partitions [BKS01].

The algorithm is similar to the basic divide-and-conquer algorithm. In the *merge* step, we need to merge all partitions such that the m-way partition fits into half the available memory and can be merged [BKS01].

Merging can also be done in a *bushy* way, i.e., merging is done in a balanced tree and each tuple is involved in $\log(m)$ merges [BKS01].

2.4.5 Index-Based Algorithms

Index-based skyline algorithms rely on the availability of indexes in the database systems. Depending on the type of the index, there are different algorithms to retrieve the skyline. Most exploit the nature of the index for quicker traversal of the data [BKS01].

Index algorithms are often desirable when the skyline is expected to be relatively small and does not contain many dimensions [BKS01].

2.4.5.1 B-Trees

B-trees can be used for two-dimensional skylines. We can go through the entire index, get the tuples in a sorted order and then apply the algorithm as described in the general two-dimensional case [BKS01].

We can also use the index to find a superset of the skyline by using the following algorithm [BKS01, Fag99]:

- Scan simultaneously through both indexes until a match is found for tuple p. A match, in this case, refers to finding the same tuple in both indexes. The first match refers to first tuple which is found in both indexes during simultaneous traversal (may not be unique).
- Any point not yet inspected is not part of the skyline since it is dominated by p.
- A point is not yet inspected if it is below p in both indexes.

For all points, retrieved as described above we can execute an arbitrary skyline algorithm to refine the superset to a skyline.

2.4.5.2 R-Trees

R-trees are a multi-dimensional indexing data structure usually used for spatial access methods which can also be used to improve the performance of skyline queries on databases. The main idea is to traverse the tree in a depth-first manner and then prune

all branches of the tree which are dominated and cannot be part of the skyline. Different heuristics can be used to select the branches of the R-tree which can have a big impact on the performance of the query. This approach requires the R-tree to contain an index on all dimensions in the `SKYLINE OF` part of the query [BKS01].

2.5 Basic Optimizations

Some optimizations for skylines can be implemented relatively systems agnostic and straightforward. We give an overview over a few of these basic optimizations in this section.

2.5.1 Pushing Skylines Through Joins

The skyline operator is usually applied after joins and group-bys but can be pushed through the joins in a limited number of cases. The cases in which the skyline operator can be pushed are defined as the *non-reductive* [CK97] joins (i.e., joins where referential integrity is ensured by foreign keys for all join predicates [CK97]). Applying the skyline before the join reduces the size of the data and makes the computation cheaper. Since non-reductive joins increase the dimensions of the processed data, the computation of the skyline operator itself also becomes less expensive [BKS01].

2.5.2 Pushing Skylines Into Joins

For queries over skylines that employ joins, it may be beneficial to compute the skyline as a part of the join itself. This is done by eliminating tuples from one side of the join which cannot be in the final skyline [BKS01].

To achieve this, both tables are sorted by the join attributes. For one or both of the tables, we also compute the skyline beforehand on the subset of dimensions not part of the join. This can be done, e.g., by sorting if only one dimension is not part of the join-dimensions (see skylines for only one dimension). In case of sort-merge-joins, this skyline computation can therefore be incorporated directly into the sorting step. The final skyline can then be computed after the join was finished using any skyline algorithm [BKS01].

This optimization technically increases the number of skylines computed by incorporating a second skyline directly into the merge step. Since the merge has to be done anyway and decreases the size of the dataset, it has the potential to achieve performance improvements [BKS01].

2.6 SkySpark

SkySpark was developed until 2016 as a framework for skyline queries in Spark [Sky]. The last version published to the official GitHub repository [Sky] is also from 2016. This

repository is also virtually the only source for SkySpark as no publications seem to exist and no documentation has been released aside from the `README` in the same GitHub repository. It uses Spark's API for retrieving the data from the database and then computes the skyline in the framework itself.

Three different algorithms are offered by SkySpark [Sky]:

- Block-nested-loop
- Sort filter skyline
- Bitmap

The implementation of the skyline queries in SkySpark does not use the Spark SQL API [Sky] but relies on the core of Spark. It interacts directly with the RDDs via the Java and Scala APIs. As such, it does not use Spark's Catalyst Optimizer. For each algorithm, there exists a different interface in SkySpark [Sky].

One of the main disadvantages of SkySpark is that it is not integrated directly into Spark. Therefore, the performance is not optimal, and it must be configured to allow proper usage with a specific instance of Spark.

According to the GitHub page [Sky], the performance of the solution has been tested in about 2500 experiments with different cluster sizes, memory sizes and data types. It claims that the solution has proved to perform well in Spark [Sky]. Sadly, the results of the tests do not seem to be available anymore. It is therefore not possible to check the results.

2.7 Distributed Algorithms for Skyline Computation using Apache Spark

Distributed Algorithms for Skyline Computation using Apache Spark [Pap18] is a Master's thesis by Ioanna Papanikolaou that explores different skyline algorithms in Apache Spark.

Among the explored algorithms is an "All Local Skyline" approach which utilizes *map* steps in Spark to first compute local skylines followed by a global skyline *merge* in main memory [Pap18]. Another algorithm is the translation of skyline queries to "plain" SQL which was already discussed above in this thesis [Pap18].

Additionally, it also features a grid-based partitioning approach which eliminates cells in the grid before the actual computation of local and global skylines [Pap18]. We will discuss this partitioning scheme in Section 3.1.3.2 and a similar algorithm in Section 3.1.4. Such approaches are not implemented as part of this thesis since we focus more on integration and grid partitioning is harder to do properly in this case. We note this approach for future work regardless.

As opposed to our work, the thesis is more about the algorithms and does not contain integration of skyline queries into Spark or Spark SQL. It also does not contain any techniques to handle incomplete datasets.

CHAPTER 3

Advanced Algorithms and Optimizations

In this chapter, we look at more advanced algorithms and optimizations for skyline queries. We limit ourselves to algorithms which satisfy at least one of the following criteria:

- The algorithm or optimization is applicable to skyline queries in Spark.
- The algorithm or optimization seems to be usable in Spark but there exist issues in practice. In this case, we will discuss those issues in Chapter 5.
- The algorithm is highly significant to skyline application(s) but not applicable to Spark.

Thus, we do not present a complete overview over algorithms and optimizations, but rather limit ourselves to "relevant" algorithms and optimizations. Any attempt at a complete list or full systematic study would vastly exceed the scope of this thesis.

We mainly focus on MapReduce algorithms in Section 3.1 followed by skyline queries on incomplete data (i.e., data containing missing values) in Section 3.2.

3.1 MapReduce Algorithms

MapReduce is a processing framework that is widely used in distributed and parallel computing. Data is first processed and turned into a key-value pair (g, x) where the key g is an identifier used for grouping while the value x holds the actual data (tuple). This is referred to as the ***map*** step in the MapReduce framework. Subsequently, the key-value pairs are ordered and distributed according to the keys in the ***shuffle*** step. For each group g, this step creates a list L_g of grouped key-value pairs. Each list of tuples is then processed independently in the ***reduce*** step which, e.g., performs the (local) skyline

computations. The output of the reducer is a new list of key-value pairs. Since MapReduce is a powerful framework, it provides a good basis for efficient skyline algorithms and has been used for skyline queries in the past [KK18, CLX+08, LZLL07, TYA+19].

In general, MapReduce approaches make use of the transitivity property of the skylines. Usually, multiple intermediate (local) skylines are computed on subsets of the data. Every tuple in the original dataset is assigned to exactly one subset for the purpose of local skyline computation [KK18]. Tuples part of a local skyline are called *skyline candidates* since they may or may not be part of the global skyline. If a point is **not** part of any local skyline, there is also no possible scenario in which it is part of the final (global) skyline. Global skylines can be retrieved by computing the skyline of all skyline candidates [KK18] or via specialized *merge* algorithms which do not require a full skyline computation [TYA+19]. We will look at an example for a merge algorithm later in Section 3.1.5.

3.1.1 Basics of Typical MapReduce Skyline Queries

The MapReduce framework is used to implement a distributed computation of skylines on potentially large datasets. It takes advantage of the fact that there are typically multiple distributed parallel workers in a MapReduce framework which can compute parts of the solution.

Computation of skylines using a MapReduce framework usually uses two stages of map, shuffle and sort, and subsequent reduction. An overview over the usual procedure can be found in Figure 3.1. The MapReduce stage 1 *partitions* the data (see Section 3.1.3 for more details) and computes a local skyline for each partition. These computations can be done in parallel since the local skylines are independent of each other. The local skylines are then used as the input to MapReduce stage 2 which computes the global skyline.

We will now look at the MapReduce algorithm involved in computing the skyline that is depicted in Figure 3.1 in more detail.

In the mapping part of the first stage, the dataset (which may be stored on multiple nodes) is split into different partitions (also referred to as *groups*) such that every data point is assigned to exactly one partition. Partitioning schemes will be discussed later in Section 3.1.3 such that we will not go into detail here. Every partition P has an assigned unique partition identifier p which is used by the MapReduce framework. For each tuple t_x in the dataset and the identifier p of its assigned partition, a key-value pair (p, t_x) is produced. Data for each partition is then sorted and shuffled such that each reducer receives a list of key-value pairs L_p associated to a single partition P. The partition identifier p is stored in the key while the value portion t_x contains the tuple. Since each reducer receives all key-value pairs (p, t_x) with the same key p, this ensures that the data from each partition is sent to the correct reducer. The reducer calculates the local skyline from the given subset of tuples stored in the value portion t_x of the pairs (p, t_x). Each skyline candidate is then part of the output of the reducer and used for further processing. All data points eliminated during the local skyline computation

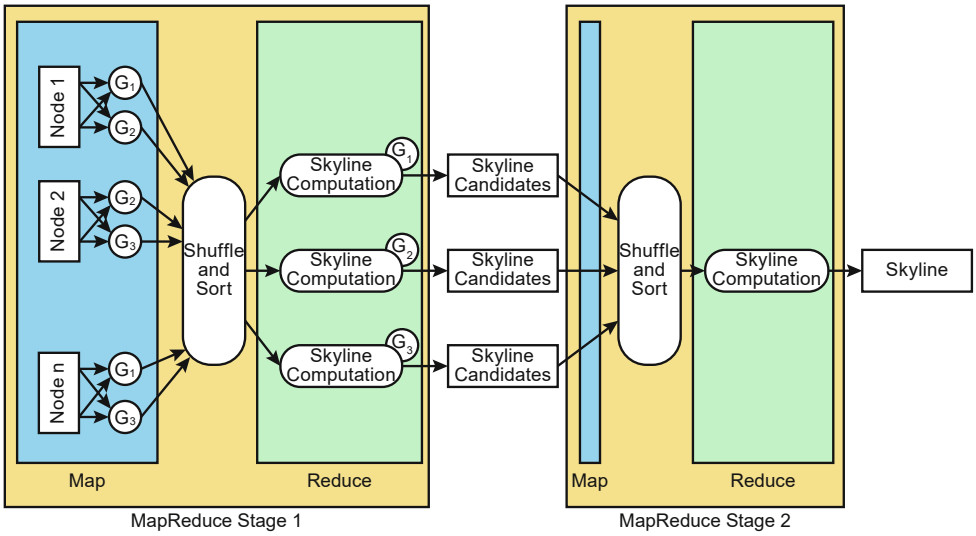

Figure 3.1: MapReduce Skyline Query Processing

cannot be part of the global skyline due to the transitivity property of skylines (see Chapter 2). They are therefore not part of the output of the reducer. Since the set of skyline candidates is usually significantly smaller than the original dataset, this approach usually reduces the input size of the global skyline computation significantly.

In the subsequent second MapReduce stage, the skyline candidates are again mapped, shuffled, and sorted. They are used in another reduce step to compute the global skyline. Since the mapping step of the second MapReduce stage is usually trivial, it is only hinted at in Figure 3.1. The global skyline can easily be computed by a single reducer which receives all skyline candidates as input data and computes the global skyline using an arbitrary skyline algorithm. This approach has the drawback that it, while being simple, can only run on a single node. Parallelization and simplifications are introduced by the advanced algorithms which we will discuss later. For example, in Section 3.1.5 we will discuss an approach that uses a simpler "merge" algorithm instead of a full skyline computation.

The first MapReduce stage can be repeated multiple times, e.g., with different partitions due to the transitivity property of the skyline. If a point is eliminated in any local skyline, it cannot be part of the global skyline. While this approach may further decrease the input size of the global skyline computation, it can also introduce additional processing costs for the local skylines as well as increased communication costs.

While the general procedure for MapReduce often follows the schema outlined above and depicted in Figure 3.1, there are often optimizations including:

- Preprocessing (including but not limited to presorting and indexing)
- Partitioning of the data
- Shuffling and assignment of data to skyline computation
- Algorithm used for the local skylines
- Number of local skyline steps before global skyline computation
- Number of parallel processing nodes (e.g., for local skylines)
- Algorithm used for computing the global skyline

3.1.2 Reducing Processing Costs, Communication Costs, and Overhead

Optimizations of MapReduce algorithms usually involve optimized partitioning (see Section 3.1.3) of the data such that the results of the intermediate processing steps are as small as possible. The smaller the local skylines, the lower the communication costs (e.g., between the stages; see Figure 3.1) and the faster the global skyline can be computed [KK18, TYA+19]. To achieve this, most approaches also take the concrete values of the dataset into account for the distribution. This is part of the partitioning of the data [KK18] which will be discussed in detail shortly.

Reducing the overhead induced by parallelization of algorithms is also vital for fast and efficient computation of the skyline. Overhead is usually caused by the additional work needed for synchronization and computation of the global skyline from the intermediate results [TYA+19].

Increased overhead and communication costs often stem from insufficient partitioning schemes which do not properly take the data into account. To speed up computation, it is necessary to both distribute the work over the parallel workers as evenly as possible and keep the local skylines as small as possible. While an even distribution will reduce the probability of stragglers and skew, smaller local skylines will reduce both the communication and processing costs of the global skyline.

Since it is now clear that the partitioning of the data is vital to the algorithms, we will now discuss some useful partitioning schemes.

3.1.3 Partitioning Schemes

How the data is partitioned is usually defined by a scheme. The exact scheme employed is subject to each concrete algorithm, but there are three general kinds of partitioning as outlined in [KK18].

3.1.3.1 Random Partitioning

Random partitioning distributes the data into partitions at random where partitions ideally are equal in size. The size of the resulting skylines per partition is expected to be equal since the samples are random for each partition. It is one of the simplest partitioning methods and introduces virtually no overhead [KK18].

One main disadvantage of this approach is that the total number of skyline candidates is not minimized. The local skylines usually contain many random tuples not part of the global skyline since they are dominated by tuples in other partitions. This increases the communication costs between the nodes and the costs for computing the global skyline [KK18]. It may be more beneficial to chose partitions such that the number of eliminated tuples in the local skyline is maximized.

3.1.3.2 Grid-Based Partitioning

Grid-based partitioning is the most common kind of partitioning in skyline computation [KK18]. Each dimension is divided into m parts such that there is a total of $m^{|D|}$ partitions in the grid where D is the set of dimensions relevant to the skyline. This method of partitioning can be used to *prune* (i.e., eliminate) partitions which cannot be part of the skyline in a filtering step [KK18].

The data space (i.e., range) for a partition is represented by [KK18]:

$$R_{P_i} = [l_1^{i-1}, l_1^i] \times \cdots \times [l_{|D|}^{i-1}, l_{|D|}^i]$$

where l_j^{i-1} and l_j^i are the boundaries of the jth dimension for the ith partition.

Grid partitions at the "corners" are more likely to contribute to the skyline than others [KK18]. It is not defined whether the grid based partitioning uses an equi-width (equal width for each partition in a dimension) or an equi-depth (equal number of elements per partition in a dimension) approach. Potential optimizations may therefore be achieved by carefully choosing the width of the partitions for each dimension.

3.1.3.3 Angle-Based Partitioning

Angle-based partitioning maps the Cartesian coordinates of a tuple to coordinates in a *hypersphere*. We will not go into the (mathematical) details of this method since it is much more complex than the others. Instead, we only give the basic formulas and definitions since it is a common partitioning scheme.

The tuples of Cartesian coordinates $p = (p_1, p_2, \ldots, p_{|D|})$, where D is the set of dimensions relevant to the skyline, are mapped to *hyperspherical coordinates* which consist of $|D| - 1$ angular coordinates $\Phi_1, \Phi_2, \ldots, \Phi_{|D|-1}$ [KK18].

The data space can then be represented by $R_{P_i} = [\Phi_1^{i-1}, \Phi_1^i] \times \cdots \times [\Phi_{|D|-1}^{i-1}, \Phi_{|D|-1}^i]$ $(1 \leq i \leq N)$ where $\Phi_j^0 = 0$ and $\Phi_j^N = \frac{\pi}{2}$ $(1 \leq j \leq |D|)$ and N is the number of partitions [KK18].

While referring to [VDK08] for definitions and theoretical foundations, we now give the formulas taken from [VDK08] for a tuple $x = (x_1, x_2, \dots x_{|D|})$ (for the dimensions D relevant to the skyline computation) and N partitions. Here, the coordinates consist of a radial coordinate r and $|D| - 1$ angular coordinates $\Phi_1, \dots \Phi_{|D|-1}$ [VDK08]. To keep the formulas more readable, we write n instead of $|D|$ which means that $n = |D|$.

$$
\begin{aligned}
r &= \sqrt{x_n^2 + x_{n-1}^2 + \dots + x_1^2} \\
tan(\Phi_1) &= \frac{\sqrt{x_n^2 + x_{n-1}^2 + \dots + x_2^2}}{x_1} \\
&\dots \\
tan(\Phi_{n-2}) &= \frac{\sqrt{x_n^2 + x_{n-1}^2}}{x_{n-2}} \\
tan(\Phi_{n-1}) &= \frac{x_n}{x_{n-1}}
\end{aligned}
$$

We note that, in general, $0 \leq \Phi_i \leq \pi$ for all $1 \leq i \leq |D| - 1$ and $0 \leq \Phi_{|D|-1} \leq 2\pi$ [VDK08]. Under the assumption that $\forall i : x_i \geq 0$ (w.l.o.g.) it holds that $0 \leq \Phi_i \leq \frac{\pi}{2}$ for $1 \leq i \leq |D| - 1$ [VDK08].

The partitions can then be chosen based on the hyperspherical coordinates. A common approach is to partition the space in such a way that each partition contains (roughly) the same number of data points. Partitions can also be chosen dynamically such that first all data are assigned to the same partition which is then gradually split until a desired number of partitions with roughly the same number of elements is reached. This has the effect that some areas of the data space are more finely partitioned than others. Details on how to compute the partitions can be found in [VDK08].

3.1.4 An Efficient Parallel Processing Method for Skyline Queries in MapReduce

In this section, we discuss the implementation of skyline algorithms based on a MapReduce framework presented in [KK18]. It is a parallel and distributed algorithm suited for large amounts of data. The algorithm follows the basic MapReduce approach described in Section 3.1.1 where an overview can be found in Figure 3.1. It mainly relies on grid-based partitioning and the application of filtering techniques to reduce the effort needed and reduce the overhead.

The main difference is that it employs two filters in addition to the skyline computation performed by the reducers. *Outer-cell filtering* is applied when computing the local skylines while *inner-cell filtering* is used to speed up the computation of the global skyline [KK18]. Partitions are computed using grid-based partitioning already described above. The partitions in the partition grid are also referred to as *cells*.

Since the algorithm is mostly based on the filtering strategies for local skylines (outer-cell filtering) and the global skyline (inner-cell filtering), we now describe both filtering approaches.

3.1.4.1 Outer-Cell Filtering

Outer-cell filtering removes all non-empty cells for which all tuples are dominated by tuples in other cells. Such *unqualified* cells are defined by a new variant of the dominance relationship which describes dominance between entire cells instead of single tuples [KK18].

For each cell C_i, we define the best point bp as the corner point that has the best values in every dimension of the cell. Similarly, the worst point wp is defined as the corner point with the worst values in every dimension of the cell. We write $C_i.wp$ and $C_i.bp$ for the worst and best point of cell C_i respectively and use $C_i.wp_j$ and $C_i.bp_j$ to refer to the value of the point in dimension d_j.

From this point on, we assume, for simplicity of notation, that all dimensions are minimized. The best point $C_i.bp$ for cell C_i is therefore closest to the origin while the worst point $C_i.wp$ is the farthest away. All definitions, observations, and algorithms also hold analogously for maximization.

The range of cell C_i is defined as:

$$[C_i.bp_1, C_i.wp_1] \times [C_i.bp_2, C_i.wp_2] \times \cdots \times [C_i.bp_{|D|}, C_i.wp_{|D|}]$$

where D is the set of dimensions relevant to the skyline computation [KK18].

Based on these definitions, we now give the definition of the *total dominance relationship* under the assumption of minimization of all dimensions:

Definition 6. *Total dominance relationship [KK18]*

A cell C_i totally dominates a cell C_j if and only if $C_i.bp_k < C_j.bp_k$ for all $d_k \in D$ where D is the set of dimensions relevant to the skyline computation. We denote this by $C_i \succ_T C_j$.

From this, we can derive the following lemma:

Lemma 1. *[KK18] For two tuples p and q such that p is part of the non-empty cell C_i and q is part of C_j where $i \neq j$ it holds that if $C_i \succ_T C_j$, then q is not part of the global skyline.*

Based on the total dominance relationship, we can now define the outer cell filter.

Definition 7. *Outer-cell filter [KK18]*

For each cell C_i, the outer-cell filter is defined as $C_i.wp$ We denote this filter point $\mathrm{OCF}(C_i)$*. The point* $\mathrm{OCF}(C_i)$ *is maintained for every non-empty cell. Using this, we can filter out tuples dominated by the cell.*

3.1.4.2 Inner-Cell Filtering

The main purpose of the inner-cell filtering is to be able to pick points in the global skyline from the local skyline in a parallel manner. This allows for efficient computation of the skyline using the MapReduce framework.

We define the *partial dominance relationship* under the assumption that all dimensions are minimized.

Definition 8. *Partial dominance relationship [KK18]*

A cell C_i partially dominates a cell C_j where $i \neq j$ in a skyline with the relevant dimensions D if and only if:

- $C_i.bp_k \leq C_j.bp_k$ *for every* $1 \leq k \leq |D|$
- $C_i.bp_k = C_j.bp_k$ *for at least one* $1 \leq k \leq |D|$
- $C_i.bp_k < C_j.bp_k$ *for at least one* $1 \leq k \leq |D|$

We denote this by $C_i \succ_P C_j$.

From this, we then define partially dominated and partially anti-dominated cells.

Definition 9. *Partially dominated cells [KK18]*

The partially dominated cells of C_i are the cells C_j for which $C_i \succ_P C_j$ holds. We denote the set by $\mathrm{PD}(C_i)$.

Definition 10. *Partially anti-dominated cells [KK18]*

The partially anti-dominated cells of C_i are the cells C_j for which $C_j \succ_P C_i$ holds. We denote the set by $\mathrm{AD}(C_i)$.

We observe that for any given two cells C_i and C_j where $i \neq j$ it holds that $C_i \in \mathrm{PD}(C_j)$ if and only if $C_j \in \mathrm{AD}(C_i)$. We now give a lemma about the dominance relationship between two tuples in different cells for which a proof can be found in [KK18].

Lemma 2. *[KK18] For two tuples p and q such that p is part of C_i and q is part of C_j where $i \neq j$ it holds that if $p \succ q$ then $C_i \succ_T C_j$* ***or*** *$C_i \succ_P C_j$.*

From this, it intuitively follows that for each tuple q in C_j it suffices to check tuples in cells C_i such that $C_i \succ_T C_j$ or $C_i \succ_P C_j$ to determine whether q belongs to the global skyline. Since tuples in totally dominated cells are already eliminated in the local skyline filtering, it suffices to only check tuples in C_i such that $C_i \succ_P C_j$. This leads us to the following theorem (proof available in [KK18]):

Theorem 1. *[KK18] Given a local skyline of cell C_j, the tuple q in the local skyline is part of the global skyline if q is not dominated by any tuples in the local skylines of all cells C_i such that $C_i \in \mathrm{AD}(C_j)$ (i.e., $C_i \succ_P C_j$).*

The *inner-cell filter* is then the filter which filters out any tuples not part of the global skyline.

Definition 11. *Inner-cell filter [KK18]*

The inner-cell filter of cell C_j is the set of local skylines in all cells C_i such that $C_i \in \mathrm{AD}(C_j)$. We denote it as $\mathrm{ICF}(C_j)$.

It follows that only tuples in specific cells need to be compared during global skyline computations. This reduces the processing costs necessary and enables parallelism which can be achieved by handling different cells in different reducers. For parallel processing, every reducer must know all relevant partially dominating cells.

3.1.5 Distributed Skyline Query Processing Using Z-Order Space Filling Curves

In [TYA+19], Tang et al. propose an algorithm for parallel processing of skyline queries in high dimensional data environments that uses *Z-order curves*. This technique is based on [LZLL07] which first introduced the use of Z-order curves for skyline queries.

3.1.5.1 Z-Order Space Filling Curves

Z-order space filling curves [RMF+00] are used to map points from a high-dimensional to a one-dimensional space. This is achieved by drawing a "curve" which goes through all data points in the data space. We now give a high-level overview over the construction of such a curve. An example for two dimensions can be found in Figure 3.2 where the Z-order space filling curve is indicated by the dashed line.

Starting from $(0, 0)$ in the top left corner, the Z-order space filling curve can be drawn hierarchically. In the first iteration, we consider neighboring points in groups of four such that they form a square (for example, points $(0, 0), (0, 1), (1, 0)$, and $(1, 1)$). Every point is part of only a single group. The points of a group are connected by the Z-order space filling curve such that the resulting curve looks like the letter "Z". For example, the points $(0, 0), (0, 1), (1, 0)$, and $(1, 1)$ form the first group along the curve as seen in the dashed line in Figure 3.2. In the next iteration, four such neighboring groups are again grouped and connected in the shape of a "Z" by using the starts and ends of the "lower level Zs". This approach works analogously for an arbitrary number of iterations and dimensions. The dashed line in Figure 3.2 represents three iterations of this algorithm in a two-dimensional space.

Every point can be reached by traversing the curve. The order in which the tuples are encountered on the curve also gives us an ordering of the tuples which corresponds to the mapping to the one-dimensional space. Each point is assigned a one-dimensional *Z-address* which is computed by interleaving the bits of each coordinate. The Z-addresses directly correspond to the ordering of the points according to the Z-order space filling curve. If a point has a lower Z-address, it occurs on the curve before any point with

a higher address. A segment of multiple points with consecutive Z-addresses is called a *Z-region* [TYA+19]. Gaps in the addresses of the consecutive points have no bearing on the definition of Z-regions but every point in the address range must be part of the Z-region.

Definition 12. *RZ-Region [TYA+19]*

An RZ-region is the smallest rectangular area that covers a Z-region bounded by the Z-addresses of two extreme points $[\alpha, \beta]$*. The RZ-region* R *is defined by two Z-addresses* $\text{minpt}(R)$ *and* $\text{maxpt}(R)$*, i.e., the smallest Z-region that represents a rectangle containing all the data points inside the Z-region* $[\alpha, \beta]$*.*

An example for an RZ-region can be found in Figure 3.2. Depicted are four RZ-regions R_1 through R_4 with the Z-order curve represented as a dashed line. Note that the minimum and maximum points are only depicted for R_1 and define the boundary of the region.

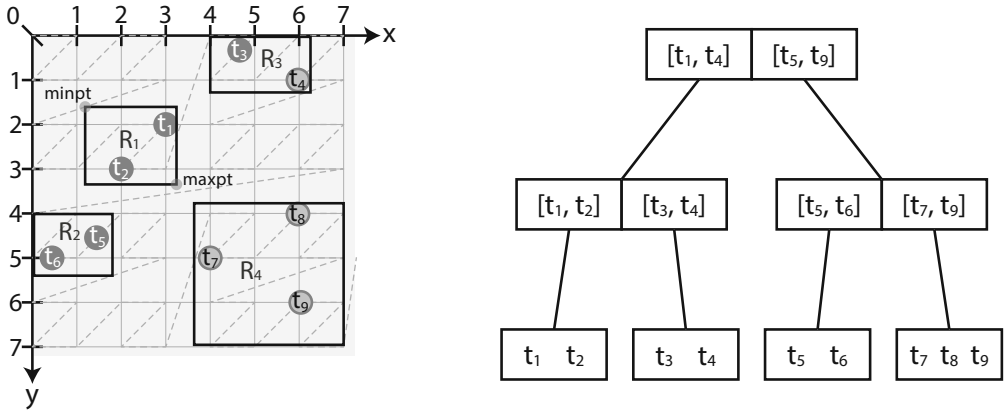

Figure 3.2: Example of RZ-region (left) and corresponding ZB-tree (right) [TYA+19]

We note that an *RZ-region* only needs to cover all data points in the Z-region but not necessarily the entire address space of the region. The Z-order space filling curve may therefore extend beyond the rectangular RZ-region as long as no data point in the Z-region is outside of the rectangle.

Additionally, we note that $[\alpha, \beta] \subseteq [\text{minpt}(R), \text{maxpt}(R)]$ and given a common prefix for α and β, **an** RZ-region can be computed by setting all bits except the prefix of $\text{minpt}(R)$ to 0 and by setting all bits except the prefix of $\text{maxpt}(R)$ to 1.

3.1.5.2 Skyline Queries using Z-Order Space Filling Curves

We can now give the following lemma regarding the dominance relationships between RZ-regions under the assumption that all dimensions relevant to the skyline are minimized:

Lemma 3. *Given two RZ-regions R_x and R_y there are three different possible dominance relationships (under the assumption that all dimensions are minimized) [TYA+19]:*

1. *If* $\text{maxpt}(R_x) \succ \text{minpt}(R_y)$ *then* $R_x \succ_T R_y$.
2. *If* $\text{minpt}(R_x) \not\succ \text{maxpt}(R_y)$ ***and*** $\text{minpt}(R_y) \not\succ \text{maxpt}(R_x)$ *then* R_x *and* R_y *are incomparable.*
3. *If* $\text{maxpt}(R_x) \not\succ \text{minpt}(R_y)$ *but* $\text{minpt}(R_x) \succ \text{maxpt}(R_y)$ *then* $R_x \succ_P R_y$.

For maximized dimensions the lemma holds analogously with minpt *and* maxpt *switched.*

Based on the RZ-regions, a *ZB-tree* can be built. It is based on a regular B-tree and hierarchically stores the data points of a skyline. All intermediate (and root) nodes in a ZB-tree correspond to RZ-regions in the data set and are denoted by their boundaries $[\alpha, \beta]$. If an intermediate note corresponding to an RZ-region has other RZ-regions as children, then those RZ-regions are contained within the parent RZ-region. We also note that all RZ-regions with the same depth in the tree are distinct, i.e., do not overlap. The data points are stored only in the leaf nodes. An example can be found in Figure 3.2. It depicts four RZ-regions (intermediate nodes) and their child tuples (leaves). All nodes corresponding to RZ-regions are children of a single root node. We note that the regions in the tree are expressed via the first and last tuple in the region when ordered by Z-order curve.

The skyline computation roughly follows the general schema of MapReduce skyline computations that was outlined above. First, the data is partitioned according to RZ-regions for which the local skylines are computed. Since, for each region, a ZB-tree can be computed, the tree can then be used to assist with the subsequent merging when combined with the observations about total and partial dominance.

We now describe the specialized merge algorithm used by this approach which replaces the full global skyline computation. The merge algorithm for computing the global skyline taken from [TYA+19] can be found in Listing 3.1.

The skyline candidates are stored in a ZB-tree. `Zsrc` contains the new data points to be added to the existing skyline `Zsky`. We traverse the ZB-tree for `Zsrc` in a breadth-first manner (similar to Breadth First Search) using a `queue` (line 4 - 5). Using the minpt and maxpt of each region, we perform dominance tests using `UDominate()` (line 6). If a RZ-region (denoted by its minpt and maxpt) is totally dominated (by `ZSky`), then it is discarded (line 7 - 8). For incomparable regions, both are part of the skyline. For this, the node is added to `dominate-branches` (line 9 - 10) to be added later (line 25 - 26). In case of partially dominated regions, the children are added to the traversal queue (line 12 - 14). Once we reach a leaf node, we need to carry out a dominance test to decide whether to discard or add the leaf node (line 16 - 19). We note that in this case every point p is also an RZ-region on its own where $p = \text{minpt} = \text{maxpt}$. The definitions for (total) dominance also hold analogously in this case (line 18). Lastly, we balance the tree using `balanceZBtree(Zsky)` (line 28) [TYA+19].

Listing 3.1: Z-merge [TYA+19]

```
input  : ZBsrc: ZB-tree for the source data, Zsky: ZB-tree for the skyline
output : Zsky: ZB-tree by merging the skyline of source data
1  queue ← empty
2  queue.enqueue(ZBsrc.root)
3  var dominate-branches
4  while queue is not empty do
5      node n ← queue.dequeue()
6      drl ← UDominate(Zsky, n.minpt, n.maxpt)
7      if ZSky dominate (n.minpt, n.maxpt) then
8          continue
9      else if ZSky incomparable (n.minpt, n.maxpt) then
10         dominate-branches.add(n)
11     else
12         if n is not leaf node then
13             for each child n' of node n do
14                 queue.enqueue(n')
15             end
16         else
17             for each child p for node n do
18                 if Zsky not dominate p then
19                     insert p into Zsky
20                 end
21             end
22         end
23     end
24 end
25 for each node n' in dominate-branches do
26     append(n', Zsky)
27 end
28 balanceZBtree(Zsky)
29 return Zsky
```

An example for merging two skylines can be found in Figure 3.3. `Zsky` is an existing skyline while `Zsrc` is a second skyline consisting of new data points to be added to `Zsky`. The result is then a new skyline which is denoted by `Zsky'`. In Figure 3.3, both the ZB-trees (right) and corresponding (hierarchical) RZ-regions can be found (left).

We use the *minpt* and *maxpt* to derive that $[t_{11}, t_{13}][t_{14}, t_{17}]$ cannot be dominated by `ZSky`. We therefore put all child nodes into the queue. We note that $\{[t_{11}, t_{12}][t_{13}, t_{13}]\}$ is partially dominated by $[t_4, t_5]$. A dominance test must be carried out for the corresponding RZ-regions. Consequently, t_{11} and t_{12} are not included in the skyline while t_{13} is. We add

the new node $[t_4, t_{13}]$ and its child to the tree. The remaining RZ-region is incomparable to `ZSky` and therefore also part of the final skyline `ZSky'`. This gives us the full merged ZB-tree which contains a valid skyline for the dataset depicted in Figure 3.3.

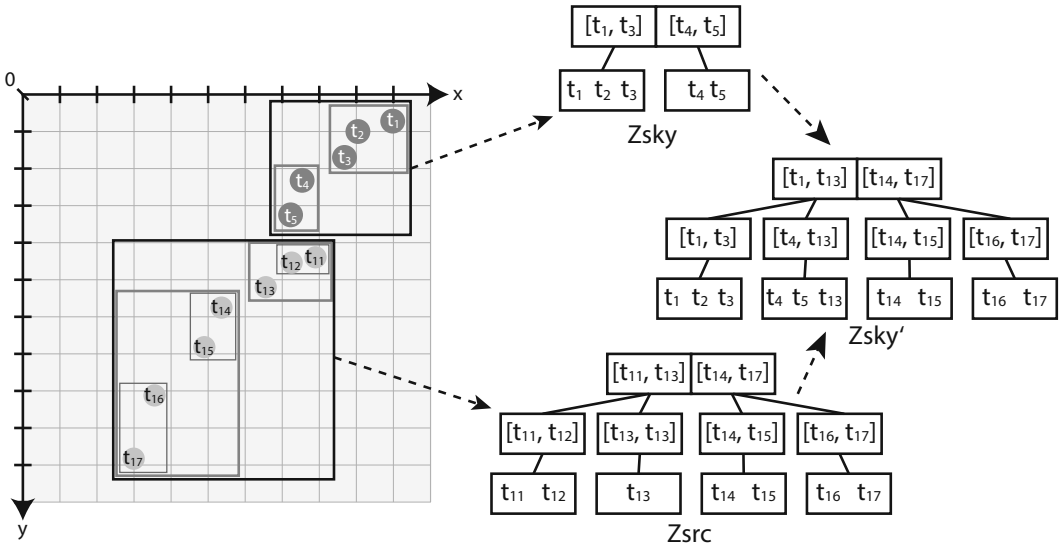

Figure 3.3: Example for Merging ZB-trees [TYA+19]

3.2 Skyline Query Processing in Incomplete Data

Up to this point, we have dealt exclusively with skyline queries on *complete* data sets, i.e., data sets where no data is missing. All definitions, observations, and algorithms up until now have assumed complete data which is crucial for the transitivity property of skylines.

3.2.1 Skylines on Incomplete Data

To outline the problems introduced by incomplete data, we consider a small example slightly adapted from [GAT19]. The placeholders "$*$" and "$\perp$" can be used to indicate missing data in the given dimension of the tuple.

We assume three tuples $a = (1, *, 10)$, $b = (3, 2, *)$, and $c = (*, 5, 3)$ and a skyline query where all three dimensions are minimized. It follows that $a \succ b$ since $1 < 3$ on the first dimension. The other dimensions are disregarded since they both have a missing value in either a or b. It similarly follows that $b \succ c$ since $2 < 5$ on the second dimension. Lastly, we note that $a \not\succ c$ since $10 \not< 3$ but at the same time $c \succ a$ since $3 < 10$ for the third dimension. This contradicts the transitivity property of the skylines on complete data because under the assumption of the transitivity property it would follow from $a \succ b$ and $b \succ c$ that $a \succ c$ which is not the case [GAT19]. Since $a \succ b$, $b \succ c$, and $c \succ a$ the dominance relationship of the three tuples is a *cyclic dominance relationship* [GAT19].

From this example, it follows that the transitivity property may be lost and *cyclic dominance relationships* are possible in incomplete data [GAT19].

Since all definitions up to this point have regarded the dataset as complete, we now give a more formal definition for incomplete datasets themselves.

Definition 13. *Incomplete Dataset [GAT19]*

A dataset P with dimensions $D = \{d_1, \ldots, d_n\}$ is incomplete if and only if there exists at least one tuple $p \in P$ that is missing data in at least one dimension $d_j \in D$ (i.e. $p_j =$ `null`).

We can now give a more formal definition of the dominance relationship on incomplete datasets which reflects the observations made above. For simplicity, we give the definition under the assumption that all skyline dimensions are minimized.

Definition 14. *Dominance Relationship on Incomplete Datasets [GAT19]*

Let P be a dataset with a set of dimensions $D = \{d_1, \ldots, d_n\}$ and let $p, q \in P$ be two points in the dataset where $p \neq q$. For a skyline with m relevant dimensions, p dominates q (written as $p \succ q$) if and only if the following three conditions hold:

- *The values of p_i and q_i for $d_i \in \{d_1, \ldots, d_m\}$ must not be missing* ***and***
- $\forall i \in \{1, \ldots, m\} : p_i \leq q_i$ ***and***
- $\exists i \in \{1, \ldots, m\} : p_i < q_i$

Based on the dominance relationship, we can now define the skyline queries on incomplete sets of data. We note that the dominance in this definition refers to dominance with regards to incomplete datasets according to the definition above.

Definition 15. *Skyline Queries on Incomplete Sets of Data [GAT19]*

The skyline $SKY_{incomplete}$ on an incomplete dataset P is built by selecting all points $p \in P$ that are at least as good as all other points $q \in P$ (where $p \neq q$) in all common non-missing dimensions and better in at least one common non-missing dimension.

We also write $SKY_{incomplete} = \{p \in R \mid \nexists q \in R \setminus \{p\} : p \prec q\}$.

The comparability between two points in an incomplete dataset is defined as follows:

Definition 16. *Comparability of Points in Incomplete Datasets [GAT19]*

In a dataset P of incomplete data, two points $p \in P$ and $q \in P$ are comparable if and only if there are no missing values for both points in at least one common dimension.

3.2.2 Translating Skyline Queries On Incomplete Data To "Plain" SQL

Similarly to the translation of queries for complete datasets introduced in Section 2.4.1, it is also possible to do the same thing for incomplete datasets. The main difference is that we must introduce handling for `null` values. All changes must be made in the `WHERE ... NOT EXISTS` part of the query which is responsible for retrieving dominating tuples in a subquery. Given our old translation procedure from Section 2.4.1, we assume that `a` is a tuple from relation `o` (outer query) while `b` is a tuple from relation `i` (inner query).

- When looking for tuples `b` which are "at least as good" as `a` we must disregard all dimensions where at least one of the values in that dimension is **`null`**. This is equivalent to checking whether the dimensions are at least as good for `b` **or** the value for `a` is `null` **or** the value for `b` is **`null`**.
- A tuple `a` can only be dominated if there exists at least one dimension in tuple `b` in which `b` is strictly better than `a` **and** the value in this dimension is **not** `null` for **both** `a` and `b`.

We now revisit the example from Section 2.4.1 again and start from the same query in skyline syntax which can also be found in Listing 3.2.

Listing 3.2: Original skyline query [BKS01, Ede09]

```
1 SELECT column_list FROM relation AS r WHERE condition(s)
2 SKYLINE OF a_1 MIN, ..., a_j MIN, a_{j+1} MAX, ..., a_k MAX, a_{k+1} DIFF, ...,
    a_m DIFF;
```

Based on these observations, we transform the query found in Listing 3.2 to a (nested) query in plain SQL. This time, we observe the particularities of incomplete datasets as listed above. While this transformation also works on complete datasets, it is inferior to the original transformation in terms of performance since it requires more conditions to be (unnecessarily) checked. The corresponding query can be found in Listing 3.3.

The lines 3 through 11 correspond to checking whether every dimensions is at least as good `OR null` for at least one of the dimensions. These conditions must hold for **all** skyline dimensions which is reflected by chaining the conditions using the `AND` keyword.

For checking whether at least one (non-`NULL`) dimension is strictly better we use the code found in line 13 through 18. **At least** one skyline dimension must be strictly better `AND` not `null`. Since it is also allowed that multiple dimensions are strictly better we can use the `OR` keyword to chain these conditions.

Listing 3.3: Translated skyline query in plain SQL [BKS01, Ede09]

```
1  SELECT column_list FROM rel AS o WHERE condition(s) AND NOT EXISTS(
2      SELECT * FROM rel AS i WHERE condition(s) AND
3          (i.a_1 ≤ o.a_1 OR i.a_1 IS NULL OR o.a_1 IS NULL) AND
4          ... AND
5          (i.a_j ≤ o.a_j OR i.a_j IS NULL OR o.a_j IS NULL) AND
6          (i.a_{j+1} ≥ o.a_{j+1} OR i.a_{j+1} IS NULL OR o.a_{j+1} IS NULL) AND
7          ... AND
8          (i.a_k ≥ o.a_k OR i.a_k IS NULL OR o.a_k IS NULL) AND
9          (i.a_{k+1} = o.a_{k+1} OR i.a_{k+1} IS NULL OR o.a_{k+1} IS NULL) AND
10         ... AND
11         (i.a_m = o.a_m OR i.a_m IS NULL OR o.a_m IS NULL)
12         AND (
13             (i.a_1 < o.a_1 AND i.a_1 IS NOT NULL AND o.a_1 IS NOT NULL) OR
14             ... OR
15             (i.a_j < o.a_j AND i.a_j IS NOT NULL AND o.a_j IS NOT NULL) OR
16             (i.a_{j+1} > o.a_{j+1} AND i.a_{j+1} IS NOT NULL AND o.a_{j+1} IS NOT
                NULL) OR
17             ... OR
18             (i.a_k > o.a_k AND i.a_k IS NOT NULL AND o.a_k IS NOT NULL)
19         )
20 );
```

3.2.3 The Optimized Incomplete Skyline Framework

In [GAT19], a framework called the *Optimized Incomplete Skyline (OIS)* is introduced. The main goal is to simplify the computation of the skyline with incomplete data by pruning data before the skyline query is processed. Recommendation systems are cited as one of the main applications for such queries [GAT19].

The algorithm is based on a framework that proposes a multi-step approach. The following steps are listed in [GAT19]:

1. Clustering
2. Sorting and Filtering
3. Local Skyline Identifier
4. Skyline Identifier

3.2.3.1 Clustering

The *clustering* step aggregates data by the comparable dimensions and plays a crucial role in preventing *cyclic dominance* and thus ensuring the *transitivity property* despite dealing with incomplete data [GAT19]. Clustering is done based on a bitmap where every dimension corresponds to one bit. If the value exists for the dimension, then the value is set to 1 and otherwise to 0. New clusters are created based on the bitmaps encountered such that each bitmap value corresponds to a cluster [GAT19]. Clustering roughly corresponds to partitioning in regular MapReduce approaches where the partitions are dictated by the properties of skylines on incomplete datasets.

For example, tuple $a = (1, *, 3)$ has a bitmap of 101 while $b = (1, 2, *)$ has a bitmap of 110. This implies that the tuples are not assigned to the same cluster and can therefore be processed independently in the next steps.

3.2.3.2 Sorting and Filtering

The *sorting and filtering* step is a preprocessing step to increase performance. During this step, two different filters are applied in order to decrease the workload of the subsequent skyline computation [GAT19].

Sorting and filtering incorporates mainly the following two optimizations [GAT19]:

- Filtering is the first action performed in this step if the user has specified a condition. For example, if the query only looks for hotels with at least three stars, all one or two star hotels are filtered away before further processing is done.
- The user can specify a *threshold* for the number of (top) data points to be retrieved for each dimension. For example, the user may specify that a threshold of 2 which means that only tuples with the top two values are retrieved for each dimension. This is achieved by first creating an ordering of tuples for each dimension and subsequently traversing this ordering. Each tuple which still fulfills the threshold is

added to the list of candidates if it was not already added before. We note that this approach takes the top **values**. If the threshold is 2 and the top value occurs twice and the second best value occurs thrice, then all five tuples are selected as part of the skyline. This optimization incorporates a user-given restriction and as such does not give any false negatives. Since only the "top" values are selected, there can also be no false positives since this skyline does not include any points not part of the "regular" skyline.

3.2.3.3 Local Skyline Identifier

The *local skyline identifier* is the local skyline computation for each candidate list of a corresponding cluster. This is done since the skyline computation on different (independent) clusters can be done in parallel and therefore sped up. The issue of cyclic dominance does not occur in this computation since all data points in a cluster have the same missing values [GAT19]. Skyline computation can simply be done by pairwise checking of the non-missing values. The result of this computation is a local skyline for each cluster which can then be used to compute a global skyline subsequently [GAT19].

3.2.3.4 Skyline Identifier

Skyline identifier is the last step and is what most other algorithms refer to as "global skyline computation". It uses the local skylines of the clusters as input and computes the final skyline.

Since we now consider tuples with different dimensions of missing data, potential cyclic dominance and the resulting loss of transitivity property have to be considered in this step. Comparable tuples can only be removed as part of the skyline computation if it can be ensured that they are, in fact, strictly worse [GAT19].

3.2.3.4.1 Proposed Algorithm from [GAT19]

In accordance with the skyline definition for incomplete data (as given above), we cannot immediately remove a dominated point since it may be part of a dominance cycle. We employ a two-step approach for each data point to detect such scenarios [GAT19].

> We go through all points in the local skyline of all clusters in order (i.e. C_1 to C_n where n is the number of clusters) and perform dominance checks. Points are skipped if they were already removed in one of the previous iterations. For each point we now apply the two-step approach:
>
> - For the current point p of the iteration we check every point q in all subsequent clusters which was not yet eliminated for dominance on the dimensions for which a value in both points exists. For example, if C_i is the current cluster, we check for all clusters C_j where $j > i$. Checking in the same cluster is not necessary since dominated points have already been removed when computing the local skyline. If dominance is detected there are two scenarios:
>   - If $p \succ q$, then q is immediately eliminated.
>   - If $q \succ p$, then a *domination flag* is set for p.
> - If after checking against all points p, the *domination flag* is set for p, then p is subsequently also eliminated from the candidate list since it is dominated by another point.

3.2.3.4.2 Counterexample

We argue that the approach to computing the global skyline given above is incorrect. To show this, we give a counterexample with simple data where applying the algorithm does not yield a correct skyline.

In this counterexample, we use the values from the example of cyclic dominance relations ship which was already introduced in Section 3.2.1. This includes the following values: $a = (1, *, 10)$, $b = (3, 2, *)$, and $c = (*, 5, 3)$.

As already discussed, we know that $a \succ b$, $b \succ c$ and $c \succ a$ in this example under the assumption that all dimensions are minimized in the skyline. From the definition of incomplete skylines and their dominance relationships, it follows that the skyline of these values must be empty as every single tuple is dominated by another tuple. We now "execute" the algorithm to check whether we get the same result.

For clustering, the tuples correspond to the bitmaps 101, 110, and 011 respectively. It follows that every tuple is in its own cluster according to the algorithm.

For the preprocessing step, none of the items are removed since we do not set a threshold or condition.

The same goes for the local skyline step since there cannot be any dominated tuples in clusters that contain only a single element.

This leaves the global skyline computation as the only step where tuples are eliminated. We now assume w.l.o.g. that a belongs to C_1, b belongs to C_2, and c belongs to C_3 as per the algorithm introduced in OIS. It follows that the algorithm starts by comparing a from C_1 to b which is the only element in C_2. Since $a \succ b$, we then remove b from C_2 after which C_2 is completely empty. Next, we go to the cluster C_3 and compare a to c. Since $c \succ a$ we set the domination flag for a. No further clusters and tuples remain from which it follows that a is deleted from C_1 since the domination flag is set. For the next iteration, we check C_2 which is now empty such that no comparisons remain for this cluster. The same also goes for the last cluster C_3 since there are no further clusters. We note that C_3 still contains c after these steps.

Cluster C_3 is the only one which still contains a tuple after the global skyline computation step and is, therefore, part of the computed global skyline. This contradicts our expected result of an empty skyline.

3.2.3.5 Global Skyline Computation

Instead of the *Skyline Identifier* which was proposed above, we can instead compare all tuples to each other and set a flag if it is dominated. By doing this, we can guarantee that all dominance relationships are found even if they are cyclic and transitivity is lost. Lastly, we can eliminate all tuples with set domination flags. This results in the final skyline.

3.2.3.6 Conclusion

Given the two-step approach of first computing the local skylines in parallel and then deriving a global skyline, this approach is very similar to the regular MapReduce framework. The main difference is that it does not apply any of the common partitioning schemes and instead partitions (clusters) in accordance to the missing values. This is necessary since otherwise parallel computations are hampered by the issues of transitivity loss and cyclic dominance. Due to the potential cyclic dominance relationships in the global skyline processing step, we cannot use the algorithm proposed in the OIS. All other ideas can be kept as-is while using a different global skyline computation as outlined above.

CHAPTER 4

Spark and Spark SQL

We discuss the basics of Spark and Spark SQL in this chapter including the parts which need to be modified or extended in order to achieve integration of skyline queries into Spark SQL.

4.1 Basic Architecture of Spark

The architecture of Spark is constantly evolving and has in parts significantly changed since the first published version. We will focus on the architecture of version 3 which is the latest version of Spark and includes significant improvements and architectural changes. Since not all sources have updated to Spark version 3 yet, we will also use sources referencing Spark version 2 where apt.

The components of Spark can be visualized as a stack. A limited overview over the stack can be found in Figure 4.1 which was inspired by [KKWZ15].

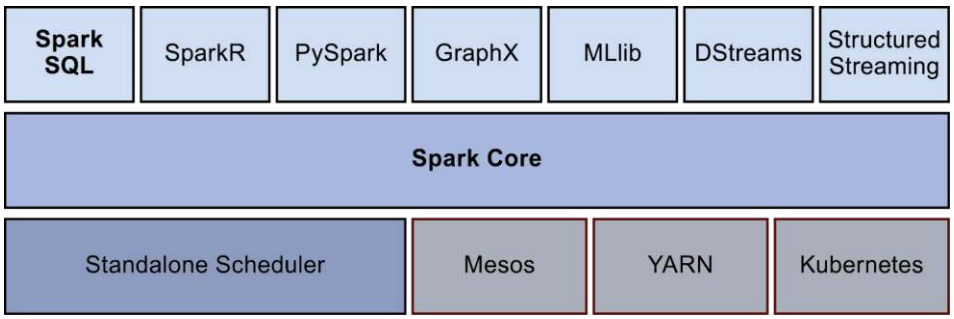

Figure 4.1: Apache Spark component stack [KKWZ15]

The central component of Spark is referred to as Spark Core and features the basic functionalities. Most other parts of Spark are extensions and based on the core. They provide additional features or support additional resources and applications.

The Spark core is responsible for task scheduling, memory management, fault recovery, storage system interaction, and many other basic tasks [KKWZ15, p. 3]. It also provides *resilient distributed datasets* (RDDs) which are the basic abstraction offered by Spark. RDDs represent collections of items which are potentially distributed across multiple nodes and can be manipulated. Users do not see the distribution but manipulations can be done in parallel. Spark offers APIs for creating and manipulation RDDs [KKWZ15, p. 3].

One of the most important extensions to the core is Spark SQL which adds SQL-like functionalities. It allows the user to write queries in a SQL-like query language. Additionally, it provides two different APIs which allow the same functionality to be expressed as function calls in a program. Those two APIs, namely DataFrame and DataSet, are often seen as equivalent but there are some differences between both APIs which will be discussed later [Spak].

A central part to the success of the Spark SQL extension is the Catalyst optimizer whose job it is to optimize queries based on predetermined rules to increase performance. This is done by manipulating the execution plan such that the result of the computations remains the same but the performance of executing the query is improved.

One of the biggest advantages of Spark SQL is the extensibility of the module. Extensions which expand the scope of the Spark SQL queries have been proposed and implemented in the past. Among those extensions there is, for example, an extension for the support of recursive queries in Spark SQL using the `WITH` keyword [GWM+19, WXG+20]. This extension is called RASQL and allows a whole new "class" of queries to be written in Spark SQL.

4.1.1 Spark Modules

Aside from Spark SQL, there are also other modules which extend the core functionality of Spark. We provide a short overview over the modules and what they are used for.

- SparkR [Spad]
  - This module provides an implementation for distributed data frames which is similar to the popular *dplyr* package for R from the *tidyverse* ecosystem [Tid]. It allows R to access functionalities from Spark such that Spark can provide an efficient data source for R programs.

- PySpark [PyS]
  - PySpark provides an interface for Spark in Python. It allows the functionality of Spark to be used in Python programs similar to the more common Java and Scala interfaces. PySpark is also one of the most common interfaces for Apache Spark.
- GraphX [Gra]
  - GraphX is a library that allows graph-parallel computations in Apache Spark. While Spark is not a graph database or graph processing engine, this module allows for graph database functionalities to be used in Spark.
- MLlib [MLl]
  - MLlib is a machine learning library for Spark that allows efficient parallel computations for distributed machine learning on Spark data sources. It features algorithms, pipelines, persistence, and other utilities for machine learning. MLlib is based on the DataFrame API and is a replacement for its predecessor ML.
- DStreams [DSt]
  - Dstreams is an extension for scalable high-throughput fault-tolerant stream processing of data streams in Spark. It is usually used in combination with streaming sources such as Apache Kafka or the Hadoop File System (HDFS).
- Structured streaming [Spah]
  - Structured streaming is a scalable fault-tolerant stream processing engine based on the Spark SQL engine. It allows incremental stream processing using the DataFrame or DataSet APIs.

4.1.2 Cluster Management Types

Spark can be run in multiple different configurations on clusters or individual machines. Performance, configuration, and maintenance of Spark instances heavily depend on which configuration Spark is run on. Usually, Spark runs on an underlying cluster management system when used in production. There are multiple management systems which are supported by the official distributions of Spark [Clu].

- Standalone
  - A simple cluster manager is included in Spark for easy cluster setup. In this mode, multiple cluster nodes may be emulated for testing purposes. The standalone mode is generally only used for development and testing and not in production.
- Apache Mesos
  - Apache Mesos is a general cluster manager that can also run Hadoop MapReduce and service applications directly.

- Apache YARN
  - Apache YARN was originally conceived as the resource manager of Hadoop 2 but can also be used to run one or more instances of Spark.
- Kubernetes
  - Kubernetes is a system mainly targeted for the purpose of automated deployment. This open-source solution also handles scaling and parameterization of the application.

4.2 Spark Version 3

We will now discuss the significant aspects of Spark version 3 and compare them to the previous versions of Spark. Version 3 was released in 2020 as an improvement over version 2 which is still supported and updated by the Apache Software Foundation. We will focus mainly on the aspects which have an impact on the integration of skyline queries into Spark SQL. This section is not a complete guide to Spark version 3 or spark in general. Since development of Spark is still ongoing, there may have been further changes to it since the writing of this thesis. The latest version of Spark as well as the documentation can be found under [Spai].

Spark version 3 runs on Java and requires a valid Java installation on the machine before it can be executed. Version 3 adds new versions of the supported software [Spaj]. Most notably this support includes [Spaj]:

- Java 11
- Hadoop 3
- Hive 3 metastore

Support for Java 11 is among the most important for the compatibility of Spark with newer systems. There is no longer a requirement to maintain an installation of Java 8 which is currently in its end-of-life phase. Adding support for Hadoop 3 and Hive 3 metastore also makes Spark compatible with a wide range of data stores which are potentially more efficient than their predecessor versions.

Spark version 3 includes some significant improvements and architectural changes for Spark SQL which are important for this thesis' goal of integrating skyline queries into Spark SQL effectively. We now list some of the most relevant changes which were taken from the official release notes of Apache Spark version 3 [Spaj].

- Enhanced ANSI SQL compatibility including a new ANSI SQL compliant mode
- Enhanced optimizer rules that increase the performance by improving the optimizations of the logical execution plan
- Calendaric improvements including a new datetime pattern definition

- Better support for correlated subquery processing
- Support for `PARTITION BY` hints in Spark SQL
- Support for `DELETE`, `UPDATE`, and `MERGE` optimizations as a part of the Catalyst optimizer
- Added `DataFrame.tail` to the Dataframe API
- New built-in functions including `sinh`, `cosh`, `tanh`, `bit_xor`, `bool_xor`

4.3 Architecture of Spark SQL

In this section, we take a look at the components of Spark SQL and provide and overview over the query execution in Spark SQL.

4.3.1 Fundamentals of Spark SQL

Spark SQL is an extension library that builds on the functionalities offered by the Spark core. Its goal is to combine the performance and functionalities of Spark with the advantages of SQL. Queries written in Spark SQL are similar to "pure" SQL and make access to structured data easier for users that are already familiar with other SQL systems.

One of the main advantages of the Spark SQL approach is that the access to the data is uniform irrespective of the data source. This allows the utilization of various different systems such as Hive or Hadoop using Spark SQL with only minimal changes if any. In case of Hive, note there is also a specialized HiveQL which is part of Hive itself and is not to be confused with any part of the Spark ecosystem. Spark also handles the distributed nature of the data which is thereby hidden from the users.

The queries can be written either as query strings or be used in programs via function calls to an API which are similar to the query strings. Results of the function calls are refined by chaining to refine the results of the queries. Every function returns an object on which other functions from the same API may be called. Both methods are translated into logical plans by Spark SQL on which the optimizations are then performed. Most queries can, in general, be expressed as both query strings and function calls via the API. None of the methods is "more powerful" than the other, but they may be more intuitive to different groups of users.

Query strings in Spark SQL have similar syntax to the syntax of other database systems. Since Spark SQL is "only" a SQL-like language, there exist queries which can only be expressed in Spark but not in pure SQL. For improved compatibility, there is an ANSI SQL compliance mode in Spark SQL which was significantly improved in version 3 of Spark [Spae]. The ANSI SQL compatibility mode can be turned on and off in Spark depending on how the system is configured.

DataFrame and DataSet both offer a different way to access the data. They are based on objects in the individual programs and provide functions which can be called to access

the functionalities of Spark SQL. The names of the methods are usually either similar to the respective SQL keywords like `select` or self-explanatory like `filter` [KKWZ15, p. 166].

DataFrames and DataSets are similar and there is often no distinction made between them which is technically not correct. The main difference is that DataSets are strongly typed (type information is known at compile time) which means that potential errors can be found at compile time as opposed to runtime with DataFrame [KW17, p. 62–63]. SQL query strings generally have the same disadvantages as DataFrames but are additionally also prone to syntax errors (including typos) which generally are only detected at runtime.

Spark SQL offers a universal optimizer which is used for all methods to express queries. This system is called the *Catalyst* optimizer and includes many rules of how a logical plan may be optimized [AXL+15]. Most rules found in the optimizer are universal, but there are some rules which are applied only for certain ways to express a query, i.e., there may be optimizer rules which are specifically applied to queries written using the DataFrame API. Such specific rules usually stem from the fact that each way to express queries has its own limitations which may influence the generated logical plan negatively. In this case, transformation rules are applied to get better results for such "typical" shortcomings.

4.3.2 Overview over Query Execution in Spark SQL

Apache Spark uses two different execution plans for executing queries. For the sake of brevity, we will sometimes use "plans" instead of "execution plans" where the meaning is clear from the context. The *logical execution plan* contains a higher-level plan that decomposes the query into smaller parts. In such a logical execution plan, joins, filters, etc. have their own node in the execution plan [KW17, p. 69]. Execution plans can be grouped in stages, where nodes within a stage may be executed independently of other stages. When the data flow crosses a stage boundary, synchronization (potentially between multiple executors) is needed in order to guarantee correct processing of the query. From now on, we will refer to all execution plans simply as *plans* where the meaning is clear.

There is a distinction between the *resolved* logical plan and the *unresolved* logical plan. In the resolved version, the names of the objects such as databases have already been resolved and replaced by the (references to) actual objects, i.e., they have already been verified [KW17, p. 69].

The *physical execution plan* is closer to the execution of the query instead of being a simple high-level graph for optimization. It contains more information about the concrete execution and algorithms used and is generally used by the layers which Spark SQL builds upon [KW17, p. 69].

We will now describe the query processing in Apache Spark SQL in more detail. An overview can be found in Figure 4.2.

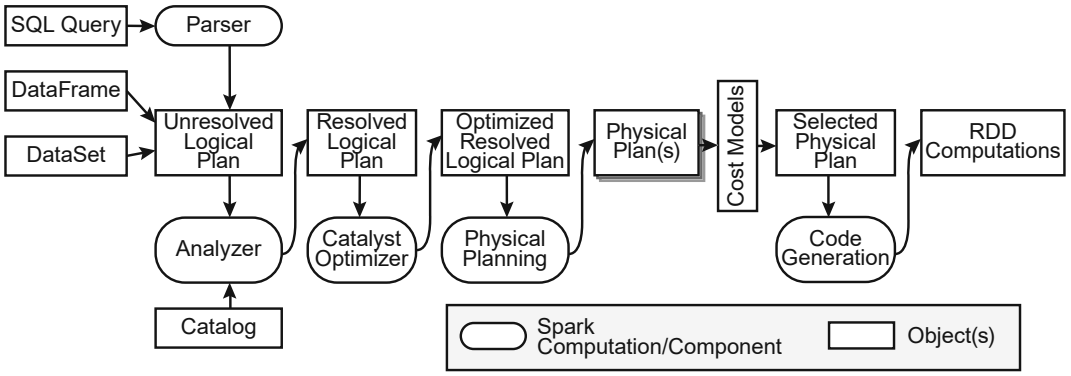

Figure 4.2: Apache Spark query processing overview [AXL+15]

For query strings, the query formulation must first be converted from its textual representation to a logical execution plan. The *Spark SQL Parser* is responsible for this step and uses matching for the string to convert parts into nodes of the logical execution plan [AXL+15]. The parser is not needed for queries that use the DataFrame or DataSet APIs since the execution plan can more directly be retrieved from the function calls.

In the next step, the unresolved logical execution plan is converted into a resolved logical execution plan by the *Spark SQL Analyzer*. The analyzer is responsible for resolving all names in the query using a metastore called the *Catalog* [AXL+15].

Optimizations are performed by the Catalyst optimizer which is one of the greatest strengths of Spark SQL. It takes a logical plan as input and transforms it via a set of rules into an optimized logical plan which can then be used for further processing [KW17, p. 69]. The rules of the Catalyst optimizer may be applied once, multiple times, or until the plan converges (up to a given threshold of iterations). Most of the rules are applicable to all queries as already mentioned above. There are some rules which apply to only a restricted set of query formulation methods, for example, only to the queries written using the DataFrame API.

After the Catalyst optimizer has finished, the logical plan is transformed into a physical plan including code generation [AXL+15] via a set of *strategies*. Those strategies and the physical plan represent the "lowest" level of Spark SQL.

Optimization of the physical plan is performed by using cost models based on which a physical execution plan is selected. Based on the selected physical execution plan, Spark generates code that uses RDDs. Further optimizations may be applied by the Tungsten optimizer which is responsible for optimizing the physical execution plan (e.g., for the given data source). This is not part of the Spark SQL system anymore but rather of the underlying core functionalities which also handle the actual query scheduling, execution, and more underlying tasks [KW17, p. 69].

4.3.3 Component Overview for Spark SQL

In this section, we will look at the components of Spark SQL in more detail. All components discussed in this section were already mentioned in the description of the query execution of Spark SQL queries.

4.3.3.1 Spark SQL Parser

The Spark SQL Parser takes the query string as an input and returns a logical plan which is yet unresolved [AXL+15]. It is used only for the input string version of Spark SQL since such functionality is not required in case of DataFrames and DataSets [AXL+15]. Since the logical execution plan generated by the parser is still unresolved, further processing steps by the analyzer are necessary [AXL+15].

The first versions used an internal parser while later ones use a derivative of ANTLR. The ANTLR tool is a lexer (tokenizer) and parser generator written in Java that allows tokenization and parsing with different target languages [ANT].

Optimization is explicitly **not** part of the lexer/parser combination in Spark SQL since it is performed by the Catalyst optimizer.

4.3.3.2 Spark SQL Analyzer

The main task of the analyzer is to take the unresolved logical plan that results from the execution of the Spark SQL parser and transform it into a resolved plan. A resolved plan differs from an unresolved insofar that names of relations and other database objects are analyzed and then matched to objects or references to objects within Spark SQL [AXL+15]. The resolved plan does not rely on the names of the objects since all data structures, types, schemata, and similar objects are already resolved. Resolving is done by the analyzer using an internal *metastorage* which is called the *Catalog* [AXL+15]. An error is raised by the analyzer if a name cannot be resolved.

From the structure of the program code and the flow of the execution plans through the Spark SQL ecosystem, the Catalyst optimizer can, in some ways, be considered a part of the analyzer. It is called by the analyzer, optimizes the resolved logical plan, and returns the (optimized) logical plan back to the analyzer. Since the Catalyst optimizer is a vital part of the success of Spark SQL, we will henceforth consider the Catalyst optimizer to be a distinct part of Spark that takes the results of the analyzer and returns an optimized plan even though this is technically passed through the analyzer.

4.3.3.3 Catalyst Optimizer

The Catalyst optimizer optimizes logical plans and returns a logical plan which is semantically equivalent but has incorporated optimizations through transformations of the execution plan [AXL+15]. Each logical plan can be displayed in a tree-like fashion. It is similar to the representation of queries in relational algebra trees which are also

often used as a basis for optimizations. Every node in the logical plan corresponds to an operation such as sort or aggregate. It governs how the query is executed on a higher logical level.

Optimizations are realized via transformations of the tree. Nodes in the tree can be replaced or moved by the Catalyst optimizer as is dictated by the set of optimization rules [AXL+15]. What optimizations can be applied is detected by pattern matching in the Catalyst optimizer. If a pattern is detected, the corresponding optimization is applied [KW17, p. 69]. Optimizations may be applied only once, multiple times, or until the logical plan converges.

Optimization rules of the Catalyst optimizer can be extended by inserting new patterns and actions for the patterns. To have full control over such modifications, a custom version of Spark must be compiled. There exist extension points in an experimental API for optimization rules but control is limited in this case. Chief among the restrictions that apply when using the extension API is that there is no control over the execution order of the optimizations. Since this often has a significant impact on the results of the optimizations, this is a significant consideration in the trade-offs when choosing how to add new rules. The usefulness of the API for the purposes of optimizations is therefore questionable.

4.3.3.4 Physical Plans and Project Tungsten

Project Tungsten is one of the most significant additions to later versions of Apache Spark. It is an accumulation of improvements in the bytecode generation and physical execution plan added in later releases. Its main purpose is to speed up the execution of the queries on the layer of the physical execution plan by generating (byte)code which can process the data from the given data source quickly and effectively [KW17, p. 50–51].

Another aim of Project Tungsten is to reduce the memory footprint of queries by optimizing the code compilation to prevent memory explosion. This can be achieved by intelligent cache-aware computations which revisit previous computations saved in the cache. Advanced cache handling techniques are one of the important parts of the success of Spark since they allow handling of big datasets even with the limited physical memory available [KW17, p. 51].

4.3.4 Extending Spark SQL

In this section we go over the various possibilities of extending Spark SQL as well as their advantages and disadvantages. We also give a verdict of which method is best suited for integrating skyline queries into Spark SQL.

4.3.4.1 `SparkSessionExtensions` Class

The `SparkSessionExtensions` class [Spaf] is a part of the Spark API and provides injection points for the `SparkSession`. It is an experimental API as there are no guarantees about the stability or compatibility made by the Spark development team.

The most important extension points provided by the class include [Spaf]:

- Analyzer Rules
- Check Analysis Rules
- Optimizer Rules
- Pre-Cost-Based-Optimizer (CBO) Rules
- Planning Strategies
- Customized Parser
- Catalog Listeners (external)
- Columnar Rules
- Adaptive Query Stage Preparation Rules

The injection of extensions into the Spark Session using the `SparkSessionBuilder` has the potential to be a much easier and potentially more sustainable way of extending Spark SQL but also has drawbacks.

The first drawback is due to the experimental nature of the developer API. Since there are no guarantees that the API will continue to work in future versions, compatibility with even minor updates may not be given [Spaf].

Another drawback is that the restricted control when using the API. For example, when extending the optimizer rules, new rules can be added with relative ease but the order of the application of the optimizer rules cannot be changed [Spaf]. Since the results of the optimizations may vary greatly depending on the order, the performance when using the extension API may be insufficient.

Also, not all algorithms can be implemented using the API as some of them may require changes that go beyond the functionalities mentioned above. For example, it is not possible to write custom RDDs and other lower-level functionalities using the API.

4.3.4.2 Extending the Source Code

Extending the source code has the drawback that all changes must be compiled into a new binary of Spark. This takes more time and provides potentially more points of failure than simply using an API.

On the other hand, this method allows full control and does not restrict what can be implemented. It is possible to tweak the order of optimizations and lower level functionality can also be implemented.

If the changes are supposed to be a permanent part of Spark, then this method is also preferable since all changes can be simply pushed to the GIT repository and merged as needed. This makes it theoretically possible to include the skyline functionality in the main Spark releases.

4.3.4.3 Conclusion for Extending Spark

Overall, the API only seems to be really useful for quick debugging and experimenting but does not provide the support and functionality necessary for long-term sustainable development of the required features for integrating skyline queries into Spark SQL.

Consequently, we will extend the source code directly as part of this thesis.

4.4 Integration Steps for Skyline Queries into Spark SQL

In this section, we discuss which parts of Spark in general and Spark SQL in particular need to be modified to allow Skyline queries to be integrated into Spark SQL.

4.4.1 Modifications of Spark SQL Parser

One of the main components that must be modified is the Spark SQL parser such that the skyline queries can be expressed as Spark SQL query strings. The syntax of skyline queries in SQL has been defined since the inception of skyline queries in [BKS01]. Modifications of the parser part of Spark SQL are universal since the basic (non-optimized) logical plan is straightforward and does not require modification for different implementations of the queries. We note that this part is distinct from the analyzer that resolves the plan and the Catalyst optimizer that transforms the non-optimized plan into an optimized plan.

4.4.2 Modification of Spark SQL Analyzer

For basic functionality, the analyzer does not require significant adaptations since the resolution of names is similar to the resolution in non-skyline queries in Spark SQL. Changes are only needed for some special cases for resolutions where the skyline query is paired with an aggregate. In this case, special handling of the following is necessary:

- Resolution of aggregation attributes (`min`, `sum`, ...).
- Resolution of attributes through `filter` generated by `HAVING`.
- Preventing premature projections which by default are inserted after the aggregate or after `HAVING` but must be moved to after the skyline.

4.4.3 Modification of the Catalyst Optimizer

Most existing optimizations of the Catalyst optimizer can be left as-is for the skyline queries since most parts of the skyline query are the same as in a regular query. This

follows from the fact that every skyline query is an adapted `SELECT-FROM-WHERE` statement which means that most optimizations still apply. For the newly introduced nodes in the logical execution plan, new optimization rules must be devised. The old rules do not cover any optimization specific to skyline queries. Additionally, the optimizations for skyline queries depend (in part) on which algorithm is used.

4.4.4 Execution and Physical Plan

Since most algorithms cannot be implemented using pure SQL or Spark SQL, additional code is necessary to ensure that the query is executed according to the execution plan quickly and effectively. This includes changes to the bytecode generation and to-be executed code.

Modifications to the underlying data structures and mechanisms of Spark may be necessary to increase performance or enable specific algorithms. This is comparable to the introduction of recursive queries to SQL as done in RASQL also necessitated the introduction of a new RDD to efficiently implement the algorithm [GWM+19, WXG+20].

CHAPTER 5

Integrating Skyline Queries into Spark SQL

In this chapter, we discuss the integration of skyline queries into Spark SQL. This includes query string parsing, extending the DataFrame and DataSet APIs, and adapting query plans in Spark SQL to optimize the support of skyline queries. We will also look at how the various algorithms can be implemented based on Spark SQL.

This chapter contains actual code from the implementation instead of the pseudo code that was used in previous chapters. In some cases, we only look at snippets and shortened versions of code since the full code can be extensive. To make the code more readable, we use `[...]` to mark sections of code where a (significant) portion was cut.

5.1 Using the Integration of Skyline Queries into Apache Spark SQL

Before proceeding to the actual implementation, we first look at how the integration will be used. This makes it easier to understand the rest of this chapter. All functions and algorithms used in this section will be explained in more detail in later sections of this chapter.

5.1.1 Query Strings

When coming from systems that support SQL or SQL-like languages, the query strings are the most intuitive method to formulate skyline queries. We use the existing Spark SQL data retrieval syntax (found in Listing 5.1) as a basis for our implementation.

The contents of the placeholder `select_statement` from Listing 5.1 can be found in Listing 5.2. The latter is also the part that was modified to allow skyline queries.

Listing 5.1: Data retrieval query in Spark SQL

```
[ WITH with_query [ , ... ] ]
select_statement [
  { UNION | INTERSECT | EXCEPT }
  [ ALL | DISTINCT ] select_statement, ...
]
    [ ORDER BY { expression [ ASC | DESC ]
      [ NULLS { FIRST | LAST } ] [ , ... ] }
    ]
    [ SORT BY { expression [ ASC | DESC ]
      [ NULLS { FIRST | LAST } ] [ , ... ] }
    ]
    [ CLUSTER BY { expression [ , ... ] } ]
    [ DISTRIBUTE BY { expression [, ... ] } ]
    [ WINDOW { named_window [ , WINDOW named_window, ... ] } ]
    [ LIMIT { ALL | expression } ]
```

Listing 5.2: Extended Spark SQL retrieval query syntax including skyline queries

```
SELECT [ hints , ... ] [ ALL | DISTINCT ] {
[ [ named_expression | regex_column_names ] [ , ... ]
    | TRANSFORM (...) ]
}
    FROM { from_item [ , ... ] }
    [ PIVOT clause ]
    [ LATERAL VIEW clause ] [ ... ]
    [ WHERE boolean_expression ]
    [ GROUP BY expression [ , ... ] ]
    [ HAVING boolean_expression ]
    [ SKYLINE OF [ DISTINCT ] [ COMPLETE ]
      { expression { MIN | MAX | DIFF } [ , ... ] } ]
```

We can see that the *skyline clause* (part starting with `SKYLINE OF`) must come **after** the `HAVING` clause but **before** any parts of the query organization (which, for example, includes `SORT` and `ORDER`).

For our implementation, we allow the use of the keyword `COMPLETE` which forces Spark to use a complete skyline algorithm instead of an incomplete. It must come after `SKYLINE OF` and `DISTINCT` (if applicable) but before the skyline dimensions. The keyword `BNL` can also be used instead of `COMPLETE` to force a (complete, non-distributed) block-nested-loop algorithm. Since this is only used for debugging, testing, and benchmarking purposes, it was omitted here.

Since the query strings are universal to all APIs, they can be used everywhere as long as a `sql()` function exists. This includes the Scala API where the query can be written as follows:

```
sqlcontext.sql("SELECT * FROM hotels SKYLINE OF price MIN,
distance MIN").show
```

In Python (PySpark), the syntax is nearly identical:

```
sqlContext.sql("SELECT * FROM hotels SKYLINE OF price MIN,
distance MIN").show()
```

The query strings allow an arbitrary number of dimensions as well as both distinct and non-distinct skyline queries. They also allow `MIN`, `MAX`, and `DIFF` to be used on any dimension.

Additionally, we note that, while such queries are valid in other APIs, the query strings **do not** allow skyline queries without dimensions. The skyline operator can simply be omitted in this case since the result of such a skyline query is the input data set.

5.1.2 Scala DataFrame and DataSet API

The DataFrame and DataSet API provides a method of formulating skyline queries via function calls. Syntactically, the integration of skyline queries is similar to all other functions provided by the APIs. In the following code and examples we limit ourselves to Scala code since it is most commonly used, e.g., in the Spark shell.

Skyline dimensions can be specified via tuples of two strings. The first string is the column (or expression) of the dimension while the second specifies whether minimization, maximization or difference is desired. For a non-distinct skyline, we use the function `skyline` in the API. An example for a simple skyline query with two dimensions may look like this:

```
df.skyline(("price", "min"), ("distance", "min")).show
```

For a distinct skyline, the example is similar but instead of `skyline`, the function `skylineDistinct` is used:

```
df.skylineDistinct(("price", "min"), ("distance", "min")).show
```

Similarly, we can also pass the skyline dimensions via the columns. In this case, we use the methods `smin`, `smax`, and `sdiff` on the columns. One example for a non-distinct skyline can be as follows:

```
df.skyline(col("price").smin, col("distance").smin).show
```

The syntax for a distinct skyline is again similar:

```
df.skylineDistinct(col("price").smin, col("distance").smin).show
```

5.1.3 PySpark DataFrame and DataSet API

The integration into PySpark is similar to the Scala/Java integration which it is based on. Since tuples do not exist in Python in the same way as in Scala, we use lists of strings instead. The first list contains the dimensions/columns while the second one, named `minMaxDiff`, contains the types of skyline dimension (e.g., minimization). Both lists of strings are used to generate pairs which represent the dimensions. The pair of first entries in the lists together represent the first dimension and so on. It follows that the order of elements in the lists is crucial. An example for a non-distinct skyline query using `skyline` with two minimized dimensions is as follows:

```
df.skyline(
  ["price", "distance"],
  minMaxDiff=["min", "min"]
).show()
```

For a distinct skyline, we simply use `skylineDistinct` instead of `skyline`:

```
df.skylineDistinct(
  ["price", "distance"],
  minMaxDiff=["min", "min"]
).show()
```

Alternatively, we can also use the columns and use the functions `smin()`, `smax()`, and `sdiff()` to write the query. An example for a non-distinct query is as follows:

```
df.skyline(df.price.smin(), df.distance.smin())
```

The distinct variant is, again, similar:

```
df.skylineDistinct(df.price.smin(), df.distance.smin())
```

5.2 Source Code Modification and Extension API

The advantages and disadvantages of direct source code modification versus using the (experimental) extension API were already discussed in theory in Section 4.3.4.1. We now look at how those potential advantages and disadvantages influence the integration of skyline queries.

The main potential advantages of the extension API are faster integration and fewer changes with each version update of Spark. The main advantages of direct source code modifications are more control and potentially easier implementation of the algorithms.

For a (minimally viable but functional) integration of skyline queries into Spark SQL, a total of 20 files need to be modified or created. This includes support for both Spark SQL query strings and DataFrame and DataSet API in Scala, Java, and Python. It also includes support for formulating queries via *Domain Specific Language* (DSL).

We give a list of files which were modified or created as a part of the integration. For each file, we briefly describe its purpose, the necessary modifications, and how the changes likely translate to future versions of Spark (where applicable).

5.2.1 Modified Files

The files listed in this section were modified and/or extended. They therefore have the potential to cause direct conflicts when ported to future versions. We list a total of 14 modified files (counting `.py` and `.pyi` files as two distinct files).

- `SqlBase.g4`
  - This grammar file contains the ANTLR grammar for parsing Spark SQL query strings. Extending the grammar is done by adding rules.
  - This is unlikely to cause significant incompatibilities since the base grammar for Spark SQL is stable and unlikely to change in future versions.
- `catalyst/parser/AstBuilder.scala`
  - This file contains the code for traversing the ANTLR tree created by parsing a query string using the grammar specified in `SqlBase.g4`. It outputs an unresolved logical plan which can be used for further processing.
  - The *visitor* methods for each node type are based on the specified rules. Visitor methods are called during tree traversal if a node of the respective type (e.g., a skyline node) is reached. Adding a rule means that a corresponding visitor must also be added. This is expected to be as stable as the `SqlBase.g4` file.
- `catalyst/analysis/Analyzer.scala`
  - This file contains the code used for analyzing the unresolved logical plan. It outputs a resolved logical plan which can again be used for further processing.
  - The rules in this file handle the cases in which the names are resolved. For skyline queries, special rules are only necessary for some cases such as skyline queries using aggregate functions and for `filters` introduced by a `HAVING` clause.
- `catalyst/optimizer/Optimizer.scala`
  - The optimizer main class contains the central code for the Catalyst optimizer. It contains references to the rules used for optimization. Matching and rules can both be put into separate files. For details on files containing rules and matching see Section 5.2.2.
  - There is a high probability of future changes and optimizations to the existing optimizer rules. Since the changes consist only of adding references to the new rules, they can easily be reproduced.

- `core/execution/SparkStrategies.scala`
  - This file contains the strategies to convert an analyzed optimized skyline logical plan to a physical skyline plan. Both plans consist of a node and its children and descendants.
  - Depending on the number of strategies and how many choices of physical plans are available, the rules may be relatively complex. However, since the rule is simply inserted, the changes can easily be reproduced by adding the new rules. Serious compatibility problems are therefore unlikely.
- `core/Column.scala`
  - This file contains the functions referring to single columns which includes `smin`, `smax`, and `sdiff`. These functions store the respective information as part of the columns.
  - Since the changes only involve adding functions it is unlikely that there are significant incompatibilities in the future.
- `core/Dataset.scala`
  - This file contains the methods (`skyline` and `skylineDistinct`) called on DataFrames and DataSets. The inputs can either be passed via strings or columns (see `Column.scala`).
  - Since the changes only involve adding functions it is unlikely that there are significant incompatibilities in the future.
- `pyspark/sql/Column.py(i)`
  - Similarly to `Column.scala`, this file contains the functions called on single columns including `smin`, `smax`, and `sdiff`.
  - The changes only involve additional functions and future significant incompatibilities are therefore unlikely.
- `pyspark/sql/Dataframe.py(i)`
  - Similar to `DataSet.scala`, this file contains the skyline functions called on DataFrames in PySpark. The functions from Scala are called via the Py4J framework.
  - The changes only involve additional functions and future significant incompatibilities are therefore unlikely.
- `pyspark/sql/Functions.py(i)`
  - This file contains additional functionality to generate minimized, maximized, and difference columns via string representations using `smin("column")`, `smax("column")`, and `sdiff("column")` respectively. This is equivalent to calling the respective function on an existing column.
  - The changes only involve additional functions and future significant incompatibilities are therefore unlikely.

- `catalyst/dsl/package.scala`
  - This file is used for creating a Domain Specific Language (DSL) for constructing queries. It was extended to also allow skyline queries albeit not strictly necessary for minimum viable functionality.
  - The changes only involve additional functions and future significant incompatibilities are therefore unlikely.

5.2.2 Created Files

All files in the following list were created for the integration. We can omit the list of potential incompatibilities as all of them are specific to the integration of skyline queries.

- `catalyst/expressions/skyline/SkylineOperator.scala`
  - This file contains a new logical plan node that corresponds to a single skyline operator. The node contains a list of dimensions relevant to the skyline query (see `SkylineDimension.scala`) as well as a single child node. The child node provides the input data for the skyline computation.
- `catalyst/expressions/skyline/SkylineDimension.scala`
  - This file contains the information for a single skyline dimension. Multiple items are used by the skyline operator (see `SkylineOperator.scala`) to represent its dimensions. The dimension (column) is stored and evaluated as a standard Spark SQL expression. Whether the dimension is minimized, maximized, or different is indicated by a Scala case class which is comparable to an enumeration.
- `catalyst/optimizer/SkylineOptimizations.scala`
  - This file contains the optimization rules applied to skyline operators. It may contain multiple optimizations that each have an `apply()` method which uses pattern matching to identify specific skylines and then applies optimizations accordingly.
- `core/execution/skyline/BaseSkylineExec.scala`
  - This file contains the `trait` of a physical plan node for a skyline operator. For a minimally viable implementation, a single physical node that inherits this `trait` is sufficient. More advanced algorithms – especially MapReduce-like approaches – require multiple physical nodes.
- `core/execution/skyline/BlockNestedLoopSkylineExec.scala`
  - This file contains the implementation using the `trait` discussed above. We use the block-nested-loop skyline algorithm for the minimum viable integration which is reflected in the name of the file.

- `core/execution/skyline/DominanceUtils.scala`
  - This file contains utilities which can be used to check the dominance between two tuples. It is used directly by the block-nested-loop algorithm and can be reused in other implementations. Putting this functionality in a utility is not strictly necessary but helps with structuring the code and making it more reusable.

5.2.3 Conclusion about Extension versus Modification

Even with features such as PySpark integration and optimizations, the number of modified files is still small. It follows that the main potential disadvantage of source code modification, extensive necessary modifications, does not severely impede the integration. Incompatibilities with future versions are also unlikely or can be resolved quickly.

The advantages of direct source code modification like the wider range of possible features, optimizations, and algorithms by far outweigh the disadvantages. It is a better fit for properly **integrating** the skyline queries into Spark SQL.

We conclude that extending Apache Spark SQL via direct source code modification is better for our use-case than using the extension API. All algorithms presented from this point forward assume this type of integration and will pay no attention to the extension API.

5.3 Minimum Viable Integration into Spark SQL

In this section, we discuss a minimum viable integration of skyline queries into Apache Spark SQL. With these changes implemented, it is possible to write skyline queries as query strings and to use the Scala DateFrame/DataSet API. We also note that all sources represent extensions and modifications of existing Apache Spark code as of version 3 [Spai].

5.3.1 Parsing Query Strings

To integrate skyline queries into Spark SQL, we first need to extend the Spark SQL grammar that is used to parse the query strings. Spark uses ANTLR [ANT] grammar files which specify the grammar in a `.g4` file. The central parts relevant to the data retrieval statement can be found in two different rules, `fromStatementBody` and `querySpecification`. We insert a new clause `skylineClause` in both rules as seen in Listing 5.3. Note that the skyline clause is optional (indicated by `?`).

Listing 5.3: Modified Spark SQL `SELECT` statement which includes a skyline clause

```
1  fromStatementBody
2    : transformClause
3      whereClause?
4      queryOrganization
5    | selectClause
6      lateralView*
7      whereClause?
8      aggregationClause?
9      havingClause?
10     windowClause?
11     skylineClause?
12     queryOrganization
13   ;
14
15 querySpecification
16   : transformClause
17     fromClause?
18     whereClause?           #transformQuerySpecification
19   | selectClause
20     fromClause?
21     lateralView*
22     whereClause?
23     aggregationClause?
24     havingClause?
25     windowClause?
26     skylineClause?         #regularQuerySpecification
27   ;
```

After adding the skyline clause to the existing rules, we need to add two new rules to the ANTLR file. The first one, `skylineClause`, is the one which is also referenced in the modifications of the existing rules. The second one, `skylineItems`, is used as an auxiliary helper that parses the skyline dimensions individually. Both rules can be found in Listing 5.4.

The `skylineClause` starts with `SKYLINE OF` (indicated by `SKYLINE` in the rule) and is followed by an optional (indicated by `?`) `DISTINCT` and the `skylineItems` which represent the dimensions. In turn, the `skylineItems` consist of an expression which represents a column followed by exactly one of either `MIN`, `MAX`, or `DIFF`. There can be an arbitrary number of `skylineItems` per `skylineClause` but there must be at least one. If multiple `skylineItems`, are given they are separated by a comma. This is achieved by using an asterisk (`*`) to allow and arbitrary number of repetitions.

We note that `SKYLINE`, `DISTINCT`, `MIN`, `MAX`, and `DIFF` are constants which simply hold their name in string representation with the exception of `SKYLINE` which contains

Listing 5.4: Spark SQL grammar rules for a single skyline clause

```
1  skylineClause
2    : SKYLINE
3      skylineDistinct=DISTINCT?
4      skylineItems+=skylineItem (',' skylineItems+=skylineItem)*
5    ;
6
7  skylineItem
8    : skylineItemExpression=expression
9      skylineMinMaxDiff=(MIN | MAX | DIFF)
10   ;
```

the string `"SKYLINE OF"`. These tokens are necessary since ANTLR only uses single-word tokens which correspond to variables while parsing. The token's variables can then be accessed to check whether a token is present in the parsed string or not.

Every rule has a context which stores the data of the rule. Using the assignment operator =, we can tell the grammar which value we want to save in which variable. A special case is the operator += which does not store the value in the variable but rather adds it to a list of variables.

For our rules, we store the following values in the context `ctx`:

- `skylineDistinct`: This stores whether `DISTINCT` was specified in the query. If `ctx.DISTINCT != null` then `DISTINCT` was set.
- `skylineItems`: As opposed to the variable above, this holds a list of skyline items. Each skyline item also has its own context which can then be used for further processing. To simplify the syntax, we use `ctx` to denote the context of the `skylineItems`.
- `skylineItemExpression`: This holds a Spark SQL `expression` which is already defined in standard Spark SQL.
- `skylineMinMaxDiff`: This stores whether `MIN`, `MAX`, or `DIFF` were specified. As above, if, for example, the skyline dimension is minimized then `ctx.MIN != null`.

Next, we need to write visitor functions which are called by the ANTLR grammar parser if the corresponding rule is parsed. As mentioned above, there is a context associated with each rule which will be made available by the parser in the visitor functions. The visitors for both newly added rules can be found in Listing 5.5.

The first function is `withSkylineClause` which takes the context of `skylineClause` (`SkylineClauseContext`) and the logical plan node of the (single) child as parameters. We extract the `skylineItems` from the context and convert them to a Scala

list for further processing (line 5). We also extract `ctx.DISTINCT`, which is `null` if and only if `DISTINCT` was **not** specified. This is used to convert the value to either `SkylineIsDistinct` or `SkylineIsNotDistinct` (line 8 and 9). The list of `skylineItems` is mapped to `SkylineDimension` using the `visitSkylineItems` helper function which we will elaborate on shortly (line 10). The value derived from `ctx.DISTINCT`, the mapped values, and the child logical node (line 11) are used to create a new `SkylineOperator`.

In `visitSkylineItems`, we translate the `ctx.MIN`, `ctx.MAX`, and `ctx.DIFF` to the corresponding case classes using simple branching (line 17 - 23). We also use the existing expression parsing to create a new expression via `expression(ctx.expression)` (line 26). Together, this is a single skyline dimension stored in a `SkylineDimension` object (line 25 - 28).

Listing 5.5: Spark SQL ANTLR parser methods called when a skyline clause is detected

```
1  private def withSkylineClause(
2    ctx: SkylineClauseContext,
3    query: LogicalPlan): LogicalPlan = withOrigin(ctx) {
4
5    val skylineItems = ctx.skylineItems.asScala
6
7    SkylineOperator(
8      if (ctx.DISTINCT != null) { SkylineIsDistinct }
9      else { SkylineIsNotDistinct },
10     skylineItems.map(visitSkylineItems),
11     query
12   )
13 }
14
15 private def visitSkylineItems(ctx: SkylineItemContext)
16     : SkylineDimension = withOrigin(ctx) {
17   val minMaxDiff = if (ctx.MIN != null) {
18     SkylineMin
19   } else if (ctx.MAX != null) {
20     SkylineMax
21   } else if (ctx.DIFF != null) {
22     SkylineDiff
23   }
24
25   SkylineDimension(
26     child = expression(ctx.expression),
27     minMaxDiff = minMaxDiff
28   )
29 }
```

To use the skyline visitors, we also need to integrate them into the existing class as seen in Listing 5.6. Since the `skylineClause` is integrated into two different rules (see Listing 5.3), it is necessary to adapt the visitors for both rules. We note that the visitor function `withSelectQuerySpecification` (line 30 - 38) is called for both rules internally (line 1 - 15 and line 17 - 28). The actual call of the skyline visitor function is therefore handled by `withSelectQuerySpecification`.

For the `fromStatementBody`, we use the `withFromStatementBody` visitor function. In case of `querySpecification` we use `visitRegularQuerySpecification`. Note that the naming of the latter corresponds to `#regularQuerySpecification` which can be found in Listing 5.3 on line 26. The values extracted from the context are passed to the function `withSelectQuerySpecification`.

In `withSelectQuerySpecification`, we chain the calls which handle the visits of the clause. Since every single clause is optional, we need to use the function `optionalMap` (e.g., line 44 and 46). This function takes care of only calling the respective mapping function if a context exists (i.e., is not `null`). In case of the skyline clause, we use the optional map result from the preceding `withWindow` as input (line 46). The actual processing is done by the visitor function `withSkylineClause` (also line 46) as already discussed above. Consequently, we also need to change the input of the subsequent `withHints` from `withWindow` (which is now input of `withSkylineClause`) to `withSkyline` which is the output of the newly inserted (optional) processing of the skyline clause (line 47).

Listing 5.6: Spark SQL ANTLR parser visitor method calls

```
1  private def withFromStatementBody(
2    ctx: FromStatementBodyContext, plan: LogicalPlan
3  ): LogicalPlan = withOrigin(ctx) {
4    if (ctx.transformClause != null) {
5      withTransformQuerySpecification(
6        ctx, ctx.transformClause, ctx.whereClause, plan
7      )
8    } else {
9      withSelectQuerySpecification(
10       ctx, ctx.selectClause, ctx.lateralView, ctx.whereClause,
11       ctx.aggregationClause, ctx.havingClause, ctx.windowClause,
12       ctx.skylineClause, plan
13     )
14   }
15 }
16
17 override def visitRegularQuerySpecification(
18   ctx: RegularQuerySpecificationContext
19 ): LogicalPlan = withOrigin(ctx) {
20   val from = OneRowRelation().optional(ctx.fromClause) {
21     visitFromClause(ctx.fromClause)
22   }
23   withSelectQuerySpecification(
24     ctx, ctx.selectClause, ctx.lateralView, ctx.whereClause,
25     ctx.aggregationClause, ctx.havingClause, ctx.windowClause,
26     ctx.skylineClause, from
27   )
28 }
29
30 private def withSelectQuerySpecification(
31     ctx: ParserRuleContext,
32     selectClause: SelectClauseContext,
33     lateralView: java.util.List[LateralViewContext],
34     whereClause: WhereClauseContext,
35     aggregationClause: AggregationClauseContext,
36     havingClause: HavingClauseContext,
37     windowClause: WindowClauseContext,
38     skylineClause: SkylineClauseContext,
39     relation: LogicalPlan): LogicalPlan = withOrigin(ctx) {
40
41   [...]
42
43   val withWindow =
44     withDistinct.optionalMap(windowClause)(withWindowClause)
45   val withSkyline =
46     withWindow.optionalMap(skylineClause)(withSkylineClause)
47   selectClause.hints.asScala.foldRight(withSkyline)(withHints)
48 }
```

5.3.2 Logical Plan Nodes

We now take a closer look at the `SkylineOperator` node of the logical plan. It is equivalent to a single skyline operator with an arbitrary number of dimensions. A slightly shortened version can be found in Listing 5.7.

The main parts of the skyline operator node are `distinct` (line 2) and `skylineItems` (line 3). The former holds an instance of `SkylineDistinct` for which the case classes `SkylineIsDistinct` and `SkylineIsNotDistinct` exist. They are similar to enumerations since they can easily be matched via Scala pattern matching. The latter contains a Scala sequence of `SkylineDimension` which holds all dimensions associated with the skyline. Lastly, the `SkylineOperator` also contains a `child` node which is a `LogicalPlan` (line 4). It provides the input for the skyline computation.

Listing 5.7: Logical plan node representing a single skyline operator

```
1  case class SkylineOperator(
2    distinct: SkylineDistinct,
3    skylineItems: Seq[SkylineDimension],
4    child: LogicalPlan
5  ) extends UnaryNode {
6    override def output: Seq[Attribute] = child.output
7    override def outputOrdering: Seq[SortOrder]= child.outputOrdering
8    override def maxRows: Option[Long] = child.maxRows
9  }
```

The SkylineDimension contains the data for a single dimension. The dimension is stored in the form of an Expression in child (line 2) and in minMaxDiff which contains a value of the type SkylineMinMaxDiff (line 3). The possible values for SkylineMinMaxDiff are SkylineMin, SkylineMax, and SkylineDiff. This allows Scala case matching similar to distinct from SkylineOperator (see Listing 5.7). A slightly shortened version of SkylineDimension can be found in Listing 5.8.

Listing 5.8: Skyline item representing a single dimension within a skyline operator logical plan node

```
1  case class SkylineDimension(
2    child: Expression,
3    minMaxDiff: SkylineMinMaxDiff
4  ) extends Expression with Unevaluable {
5    override def toString: String = s"$child ${minMaxDiff.sql}"
6    override def sql: String = s"${child.sql} ${minMaxDiff.sql}"
7    override def nullable: Boolean = child.nullable
8    override def dataType: DataType = child.dataType
9    override def children: Seq[Expression] = Seq(child)
10 }
```

5.3.3 Analyzer

While most queries in Spark work without having to modify the analyzer, there are two changes which are necessary for special cases. All such cases are related to aggregates and HAVING clauses in the query. In this section, we will introduce changes to the analyzer which prevent these problems which will be explained in detail shortly. Alternatively, it is also possible to write all problematic queries by using subqueries and applying the skyline operator to the result of the subquery instead. From the user's perspective, the modified analyzer is preferable since the alternative requires the queries to be rewritten (slightly).

First, we ensure that we can also compute a skyline on dimensions not present in the projection (i.e., the attributes specified in the SELECT clause). For this, we expand the ResolveMissingReferences function by adding another case for the SkylineOperator. The code can be found in Listing 5.9.

The rule is applied if and only if the SkylineOperator has not yet been resolved, has missing input attributes, and the child is already fully resolved (line 1 - 2). Then we resolve the expressions and add the missing attributes (line 3 - 4) which are then used to generate a new set of skyline dimensions (line 5). If no output was added, we simply replace the skyline dimensions (line 6 - 7). Otherwise, we create a new skyline with the newly generated child (line 9) and add a projection to eliminate redundant attributes (line 10).

Listing 5.9: Analyzer extension to allow dimensions not present in the projection in the skyline operator

```
1  case s @ SkylineOperator(_, _, skylineItems, child)
2    if (!s.resolved || s.missingInput.nonEmpty) && child.resolved =>
3      val (exprs, newChild) =
4        resolveExprsAndAddMissingAttrs(skylineItems, child)
5      val dimensions = exprs.map(_.asInstanceOf[SkylineDimension])
6      if (child.output == newChild.output) {
7        s.copy(skylineItems = dimensions)
8      } else {
9        val newSkyline = s.copy(dimensions, newChild)
10       Project(child.output, newSkyline)
11     }
```

Next, we need to take care that aggregate attributes are also propagated to the skyline properly. The code to accomplish this can be found in Listing 5.10. We will only explain it briefly since it was modified from existing Spark code for similar nodes [Spai].

First, we need to find the unresolved dimensions (line 3 - 6). After this, we try to resolve the child and dimensions (line 7 - 17). We check whether we can refer to the old aggregate by reference (line 18 - 22). If such a reference already exists, we can use it in the skyline (line 23 - 44). We return the skyline as-is if nothing has changed (line 45). Otherwise, we add a new projection such that the attributes are projected away as needed (line 46 - 49).

Listing 5.10: Analyzer extension to propagate aggregate attributes to skylines

```
case skyline @ SkylineOperator(distinct, complete, skylineItems,
  aggregate: Aggregate) =>
  val unresolvedDim = skylineItems.filter { item =>
    !item.resolved ||
    !item.references.subsetOf(aggregate.outputSet) ||
    containsAggregate(item) }
  val aliasedDimensions = unresolvedDim.map(u =>
    Alias(u.child, "aggSkyline")())
  val childWithExtraDimension = aggregate.copy(
    aggregateExpressions =
      aggregate.aggregateExpressions ++ aliasedDimensions )
  val resolvedChild: Aggregate =
    executeSameContext(childWithExtraDimension)
    .asInstanceOf[Aggregate]
  val (reResolvedChildExprs, resolvedAliasDimensions) =
    resolvedChild.aggregateExpressions.splitAt(
      aggregate.aggregateExpressions.length )
  checkAnalysis(resolvedChild)
  val originalChildExprs =
    aggregate.aggregateExpressions.map(trimNonTopLevelAliases)
  val needsPushDown = ArrayBuffer.empty[NamedExpression]
  val dimensionToAlias = unresolvedDim.zip(aliasedDimensions)
  val evaluatedDimensions =
    resolvedAliasDimensions.asInstanceOf[Seq[Alias]]
        .zip(dimensionToAlias).map {
      case (evaluated, (dimension, aliasDimension)) =>
        val index = reResolvedChildExprs.indexWhere {
          case Alias(child, _) =>
            child semanticEquals evaluated.child
          case other => other semanticEquals evaluated.child }
        if (index == -1) {
          if (hasCharVarchar(evaluated)) {
            needsPushDown += aliasDimension
            dimension.copy(child = aliasDimension) }
          else {
            needsPushDown += evaluated
            dimension.copy(child = evaluated.toAttribute) } }
        else {
          dimension.copy(
            child = originalChildExprs(index).toAttribute ) } }
  val dimensionsMap = unresolvedDim.map(
    new TreeNodeRef(_)).zip(evaluatedDimensions).toMap
  val finalDimensions = skylineItems.map { item =>
    dimensionsMap.getOrElse(new TreeNodeRef(item), item) }
  if (skylineItems == finalDimensions) { skyline }
  else { Project(aggregate.output,
      SkylineOperator(distinct, complete, finalDimensions,
        aggregate.copy(aggregateExpressions =
          originalChildExprs ++ needsPushDown) ) ) }
```

In the last step, we have to eliminate the case where a projection is added automatically after the `HAVING` clause by Spark. We solve this by eliminating the projection, recomputing the references, and adding the projection back in again. This approach can also be used to fix similar problems for the `Sort` operator. The code for this analyzer step can be found in Listing 5.11. We note that we must take care to apply the rule **after** `ResolveAggregateFunctions` to make it work properly. The code for sorting and ordering is largely omitted in our code fragment since it is virtually the same and not relevant to this thesis.

We create a new rule for the logical plan which is resolved in an operators-up manner which means that children are resolved before the node itself (line 1 - 3). The rule applies if a `Project` is either between the `SkylineOperator` or `Sort` and the `Filter` introduced by `HAVING` for the `Aggregate` (line 4 and 7 - 12). It also applies if the projection was pushed between the `Sort` and the `SkylineOperator` (line 26 - 33) in which case we need to move it once more (to the outside of `Sort`).

We look at the first scenario solely based on the `SkylineOperator` (line 7 - 25). It works equivalently for `Sort` which we omit here as indicated in line 5. If the skyline was not yet resolved (line 13), we create a new logical plan which omits the `Project` and the `Filter` (line 14 - 15). This, in turn, is again a `SkylineOperator` (line 16) which can be used as input for the existing `ResolveAggregateFunctions` optimization rule (line 14) which was extended in Listing 5.10 and resolves the references in `SkylineOperator`. In the last step, we rebuild the fill logical plan including the `Project` and the `Filter` where we also take care that the projection is now the parent of the `SkylineOperator` and not vice versa (line 19 - 25).

After applying these rules, the `Project` may end up between the `Sort` and the `Skyline` in the parent-child-hierarchy or the logical plan (line 26 - 33) where the `Sort` may still be unresolved (line 34). Here, we again generate a copy of the `Sort` with the `Aggregate` as direct child (line 35 - 37) and use `ResolveAggregateFunction` to resolve it (line 35). The hierarchy can subsequently be rebuilt again with `Project` as the parent of the `Sort` (line 38 - 42).

Listing 5.11: Analyzer extension for preventing premature projections before `SkylineOperator` and `Sort`

```
1  object PreventPrematureHavingProjection extends Rule[LogicalPlan] {
2    override def apply(plan: LogicalPlan): LogicalPlan =
3        plan.resolveOperatorsUp {
4      case sort @ Sort(_, _, [...]) if !sort.resolved =>
5        [...]
6
7      case skyline @ SkylineOperator(_, _, _,
8         project @ Project(_,
9           filter @ Filter(_,
10           aggregate @ Aggregate(_, _, _)
11          )
12        )
13     ) if !skyline.resolved =>
14       val newSkylineWithoutFilter= ResolveAggregateFunctions.apply(
15         skyline.copy(child = aggregate)
16       ).children.head.asInstanceOf[SkylineOperator]
17       val newAggregate = newSkylineWithoutFilter.child
18
19       project.copy(
20         child = newSkylineWithoutFilter.copy(
21           child = filter.copy(
22             child = newAggregate
23           )
24         )
25       )
26     case sort @ Sort(_, _,
27       project @ Project(_,
28         skyline @ SkylineOperator(_, _, _,
29           Filter(_,
30             aggregate @ Aggregate(_, _, _)
31           )
32         )
33       )
34     ) if !sort.resolved =>
35       val newSortWithoutFilter = ResolveAggregateFunctions.apply(
36         sort.copy(child = aggregate)
37       ).children.head.asInstanceOf[Sort]
38       project.copy(
39         child = newSortWithoutFilter.copy(
40           child = skyline
41         )
42       )
43   }
44 }
```

5.3.4 Physical Plan Nodes

In this section, we describe what a physical plan node can look like. Afterwards, we discuss an implementation of the block-nested-loop algorithm for skyline queries (Section 5.3.4.1). Since such implementation requires checking the dominance relationship between tuples, we will introduce a utility in Section 5.3.4.2 which allows (re-)used of the algorithm for dominance checks across multiple algorithms. Lastly, we describe how to derive a physical block-nested-loop node from a given logical plan (Section 5.3.4.3). Note that we only consider complete skylines for this minimally viable integration.

We define a Scala `trait` which can be found in Listing 5.12 as basis for all physical skyline nodes and algorithms. The `trait` defines the basic properties of the skyline (line 3 - 5), skyline dimensions (line 7 - 9), input (line 13 - 14), and output (line 11 and 15 - 18) of the (physical) node.

Listing 5.12: Base `trait` for all physical skyline execution nodes

```
1  trait BaseSkylineExec extends UnaryExecNode
2  with AliasAwareOutputPartitioning {
3    def skylineDistinct: SkylineDistinct
4    def skylineDistinctBoolean: Boolean =
5      skylineDistinct == SkylineIsDistinct
6
7    def skylineDimensions: Seq[SkylineDimension]
8    def skylineDimensionExpressions: Seq[NamedExpression] =
9      skylineDimensions.map(_.child.asInstanceOf[NamedExpression])
10
11   def resultExpressions: Seq[NamedExpression]
12
13   protected def inputAttributes: Seq[Attribute] =
14     child.output
15   override def output: Seq[Attribute] =
16     resultExpressions.map(_.toAttribute)
17   override def outputExpressions: Seq[NamedExpression] =
18     resultExpressions
19 }
```

5.3.4.1 Block-Nested-Loop Algorithm

The new node for the block-nested-loop algorithms extends the `trait` introduced in Listing 5.12. It is based on the block-nested-loop skyline algorithm (see Section 2.4.3) which is fast when the entire data set scan fit in memory. A slightly shortened version of the implementation can be found in Listing 5.13. We now look at the code in more detail.

The result expression of the block-nested-loop skyline is the same as the output expression of the child (line 7 - 8). No columns are generated or removed when applying the skyline operator.

Since we only have a single global skyline operator, we have to take care of distributing all tuples such that are sent to the single node. To achieve this, we set the `requiredChildDistribution` to `AllTuples::Nil` (line 9 - 10) which puts all tuples into a single partition (on a single node). The nodes in the plan require a sequence of distributions since there may be multiple children (for example, in join nodes).

The actual algorithm is implemented in the `doExecute()` method that produces an RDD of `InternalRows` as output (line 12).

We extract some values before we start with the actual algorithm to prevent redundant computations. Since each row can consist of multiple values with (potentially) different data types, we first extract the type for each column returned by `child`. We use the sequence of expressions in `child.output` where each expression has a `dataType` field which we can use (line 13). We also store the index (ordinal) of each skyline dimension of the child output schema (line 14 - 16). This is achieved by finding the `exprId` of each skyline dimension and findings its index in the list of `exprIds` of `child.output`. Additionally, we also initialize our window (list of currently not dominated tuples) in line 17 - 18.

Next, we call `child.execute()` which returns the result of the child execution node (line 19). The function `mapPartitionsInternal` maps the partitions where each partition can be traversed via an iterator (line 19 - 20). Since we use the predefined `AllTuples` distribution scheme, there will always be a single iterator for the single partition on the single node.

We define two different indicator variables which tell us about the dominance for each row returned by the iterator. The variable `isDominated` tells us that the row was dominated by another row already in the window (line 21). We use `breakWindowCheck` to indicate that we can stop checking since the current row cannot dominate any row in the window (line 22).

Subsequently, we iterate over each row in the current window until either no element remains or `breakWindowCheck` is `true` (line 25 - 27). For each iteration, we use `DominanceUtils.checkRowDominance()` to determine whether the tuple dominates, is dominated, equal, or incomparable (line 28 - 31). The function

Listing 5.13: Minimal block-nested-loop skyline node in physical plan

```
1  case class BlockNestedLoopSkylineExec(
2    skylineDistinct: SkylineDistinct,
3    skylineDimensions: Seq[SkylineDimension],
4    requiredChildDistributionExpressions: Option[Seq[Expression]],
5    child: SparkPlan
6  ) extends BaseSkylineExec with AliasAwareOutputPartitioning {
7    override def resultExpressions: Seq[NamedExpression] =
8      child.output
9    override def requiredChildDistribution: Seq[Distribution] =
10     AllTuples :: Nil
11
12   override protected def doExecute(): RDD[InternalRow] = {
13     val childOutputSchema = child.output.map { f => f.dataType }
14     val skylineDimensionOrdinals = skylineDimensions.map{option =>
15       child.output.map{ attr => attr.exprId }
16         .indexOf(option.child.references.head.exprId) }
17     val blockNestedLoopWindow =
18       new mutable.ArrayBuffer[InternalRow]()
19     child.execute().mapPartitionsInternal { partitionIter =>
20       partitionIter.foreach { row =>
21         var isDominated = false
22         var breakWindowCheck = false
23         val dominatedSkylineTuples =
24           new mutable.ArrayBuffer[InternalRow]()
25         val iter = blockNestedLoopWindow.iterator
26         while (iter.hasNext && !breakWindowCheck) {
27           val windowRow = iter.next()
28           val dominationResult = DominanceUtils.checkRowDominance(
29             row, windowRow,
30             childOutputSchema, skylineDimensionOrdinals,
31             skylineDimensions.map { f => f.minMaxDiff } )
32           dominationResult match {
33             case Domination =>
34               dominatedSkylineTuples.append(windowRow.copy())
35             case AntiDomination =>
36               isDominated = true
37               breakWindowCheck = true
38             case Equality =>
39               if (skylineDistinctBoolean) { isDominated = true }
40               breakWindowCheck = true
41             case Incomparability | _ => // NO ACTION
42         } }
43         if (!isDominated) {
44           if (dominatedSkylineTuples.nonEmpty) {
45             blockNestedLoopWindow --= dominatedSkylineTuples }
46           blockNestedLoopWindow.append(row.copy())
47       } }
48       blockNestedLoopWindow.toIterator
49 } } }
```

`DominanceUtils.checkRowDominance()` can be found in Listing 5.14 and will be discussed later.

How each tuple is handled depends on the result of the dominance check (line 28 - 31 for dominance checks and line 32 - 41 for handling the results). Here, we use *row* when talking about a single data point (tuple) which conforms to the internal naming in Spark.

- If the current row dominates a row in the window (`Domination`), then we add the tuple to the collection of dominated tuples (line 33 -34).
- If the row is dominated (`AntiDomination`), we set `isDominated` to `true` and break from the loop which iterates over the window by setting `breakWindowCheck` to `true` (line 35 - 37).
- If the row is equal to another row (`Equality`), we again distinguish between two cases (line 38):
  - If the skyline is distinct, then we only keep one of the equal tuples and do not add the current row by setting `isDominated` to `true`. We, in this case, pretend the tuple is dominated to eliminate it and break from the iteration by setting `breakWindowCheck` to `true`. Breaking from the loop is correct in this case since an equal tuple (with regards to the skyline dimensions) is already part of the window. The current tuple cannot dominate any other tuple in the window since the window is a skyline of all tuples examined up to this point (line 39).
  - If the skyline is not distinct, we can add the tuple to the skyline since it is equal to another tuple already in the current skyline. It therefore cannot be dominated by any other tuple in the skyline and we break from the loop by setting `breakWindowCheck` to `true` (line 40).
- In case both tuples are incomparable (`Incomparabilty`), we perform no action and proceed with the comparison to the next tuple in the window unless no further tuples remain (line 41).

After the loop over the window, we add the row if and only if the row was not dominated, i.e., `isDominated` is `false` (line 43 and 46) We also remove all tuples from the window which were found to be dominated (line 44 - 45).

Finally, when all input rows in the partition have been processed, we return an iterator over the window which then holds the skyline (line 48).

5.3.4.2 Computing Dominance of Rows via Utility

We now take a look at the `DominanceUtils` and the `checkRowDominance` function which were already used in Listing 5.13. The code for the utilities can be found in Listing 5.14.

To determine the dominance relationship between two tuples (represented by rows), the following inputs are needed:

- The rows `rowA` (line 3) and `rowB` (line 4) representing a single tuple each. All computations are done "from the perspective of" `rowA`. This means, for example, if `rowA` dominates `rowB` then `Dominance` is returned and not `AntiDominance`.
- The `schema` which contains a sequence of data types (line 5). It contains **all** data types of the columns of the input rows regardless of whether the column is relevant to the skyline or not.
- The `ordinals` (line 6) are a sequence of indexes where each index identifies the index of a column relevant to the skyline query. Since the ordering of skyline dimensions does not affect the outcome of the computation, it is **not** necessary to guarantee any ordering of the `ordinals`. The assumption of ordering according to `MIN`, `MAX`, and `DIFF` assumed in some parts of this thesis for simplicity of notation does not affect the implementation.
- The sequence `minMaxDiff` (line 7) contains the info whether each skyline dimension is minimized, maximized, or different. This sequence has the same length as the sequence of `ordinals` already introduced above. It must follow the same ordering as `ordinals` with regards to the skyline dimensions since we pair the values to get the information of a single skyline dimension. For example, the first value of `ordinals` and the first value from `minMaxDiff` are used to check the first skyline dimension.

We note it follows that `ordinals` and `minMaxDiff` must contain the same number of elements while the `schema` generally contains more elements since usually not all dimensions are relevant to the skyline.

Next, we introduce three variables which help us identify which dominance case applies (line 9 - 10):

- `a_strictly_better` indicates that `rowA` is **strictly** better in at least one dimension.
- `b_strictly_better` indicates that `rowB` is **strictly** better in at least one dimension.
- `a_different_b` indicates that the values for `rowA` and `rowB` are different in at least one dimension for which the skyline type `DIFF` is used.

To compare each dimension, we iterate over the `ordinals` which we first *zip* with their indices using `zipWithIndex` (line 12). This generates a sequence of tuples that contain both the index and the ordinal. In each iteration, we have the ordinal `ord` and its index `idx` (line 13).

We use `idx` to retrieve the datatype of each column with ordinal `ord` from `schema` (line 14). Since each data type needs to be handled differently, it is necessary to have some code duplication where we repeat (nearly) the same code for each data type. Generic

code would be massively more complicated or hard to read while yielding no performance improvements. Due to the code duplication, we limit ourselves to the integer case of type `Int` or smaller (line 15 - 33). Other (omitted) cases include bigger integers (`Long`), floating point types (`Float`, `Double`), as well as decimal types (`BigDecimal`). In line 34 we indicate that such cases exist.

Another solution would be to convert all types to `Double` which has the drawback that comparisons are usually not as efficient with other data types. Due to the necessary type casting, there may also be a loss of accuracy.

If a "small" integer type is detected (line 15), we match the type more closely (line 17, 19, and 21) and use the appropriate methods for retrieving the values `valA` and `valB` for from their respective rows (line 18, 20, and 22).

The type of skyline is identified by using the index `idx` to retrieve the corresponding value from `minMaxDiff` (line 24). If either `SkylineMin` or `SkylineMax` are encountered (line 25 and 28), we check whether one retrieved values is strictly better and set the variables introduced above accordingly (line 26 - 27 and 29 - 30). In case of `SkylineDiff`, we check whether the values are different and also set the corresponding variable accordingly (line 33 - 34). We note that once one of the three indicator variables is set to `true` (condition fulfilled for at least one dimension), it is never set back to `false`.

After finishing the comparisons for each dimension, we determine the return value by checking the indicator variables:

- If the value in at least one `DIFF` dimension is actually different which is indicated by `a_different_b` being set to `true`, then the tuples are incomparable according to the skyline definition and `Incomparability` is returned (line 37).
- If either `a_strictly_better` or `b_strictly_better` is `true`, then either `Domination` (line 38 - 39) or `AntiDomination` (line 40 - 41) are returned respectively.
- If both tuples are strictly better in at least one dimension (`a_strictly_better` **and** `b_strictly_better` are true), then `Incomparability` is returned since the tuples are incomparable (line 42 - 43).
- In all other cases, since no dimension is strictly better and no `DIFF` dimension is different, the two tuples are equal and `Equality` is returned (line 44).

Listing 5.14: Utilities for checking dominance for two tuples `rowA` and `rowB`

```
object DominanceUtils extends Logging {
  def checkRowDominance(
    rowA: InternalRow,
    rowB: InternalRow,
    schema: Seq[AbstractDataType],
    ordinals: Seq[Int],
    minMaxDiff: Seq[SkylineMinMaxDiff]
  ): DominanceResult = {
    var a_strictly_better = false; var b_strictly_better = false
    var a_different_b = false

    ordinals.zipWithIndex.foreach { f =>
      val (ord, idx) = (f._1, f._2)
      schema(ord) match {
        case ByteType | ShortType | IntegerType =>
          // Integral Types (excluding Long)
          val (valA, valB) = if (schema(ord) == ByteType) {
            (rowA.getByte(ord).toInt, rowB.getByte(ord).toInt)
          } else if (schema(ord) == ShortType) {
            (rowA.getShort(ord).toInt, rowB.getShort(ord).toInt)
          } else {
            (rowA.getInt(ord), rowB.getInt(ord))
          }
          minMaxDiff(idx) match {
            case _@SkylineMin =>
              if (valA < valB) { a_strictly_better = true }
              else if (valB < valA) { b_strictly_better = true }
            case _@SkylineMax =>
              if (valA > valB) { a_strictly_better = true }
              else if (valB > valA) { b_strictly_better = true }
            case _@SkylineDiff =>
              if (valA != valB) { a_different_b = true }
          }
        [...]
      }
    }
    if (a_different_b) { Incomparability }
    else if (a_strictly_better && !b_strictly_better) {
      Domination
    } else if (b_strictly_better && !a_strictly_better) {
      AntiDomination
    } else if (a_strictly_better && b_strictly_better) {
      Incomparability
    } else { Equality }
  }
}
```

5.3.4.3 Planning the Physical Node

To make use of the skyline operator in the physical plan, we must tell Spark how to translate the skyline node in the logical plan to a valid physical plan. This is achieved by introducing a new `Strategy`. In our minimally viable case, we simply convert a single skyline node from the logical plan to a single node in the physical plan. We extend the existing `BasicOperators` strategy and add our matching there. The relevant code snippet can be found in Listing 5.15.

Listing 5.15: Query planner extension which returns a skyline physical node for a logical node

```
1  object BasicOperators extends Strategy {
2    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
3      [...]
4      case SkylineOperator(distinct, skylineDimensions, child) =>
5        execution.skyline.BlockNestedLoopSkylineExec(
6          distinct, skylineDimensions, planLater(child)
7        ) :: Nil
8      case _ => Nil
9    }
10 }
```

The strategy of the planner is defined by its `apply` function (line 2 - 9). For the logical plan node `SkylineOperator(distinct, SkylineDimensions, child)` (line 4) we create a new `BlockNestedLoopSkylineExec` node (line 5 - 7). The content of the physical plan node matches the content of the logical plan node, i.e., it also has a value for `distinct`, multiple skyline dimensions, and a single child node `child`. We use `planLater(child)` to tell the planner that the child node needs separate planning **after** the current rule was applied (line 6). This causes the planner to do another planning iteration in the future where the strategies are applied to the child.

5.3.5 DataFrame and DataSet API integration

We define two methods, `skyline()` and `skylineDistinct()` for each API integration of the skyline. The former handles skyline queries and selects all results without checking whether they are distinct while the latter handles skyline queries where the resulting tuples are distinct with regards to the skyline dimensions. We will discuss the integration into the DataFrame and DataSet APIs using Scala.

In accordance with Section 5.1, we allow two different methods for writing queries in the DataFrame and DataSet API.

The first method is using a list of tuples which each contains two strings. Column names of the skyline dimensions are the first element in each tuple while the second one specifies whether the dimension is minimized, maximized, or difference based. Valid strings for the second element in each tuple are `MIN`, `MAX`, and `DIFF`.

Listing 5.16 contains the source code for the implementation of this method. We define both `skyline` (line 1 - 9) and `skylineDistinct` (line 11 - 19) with the same inputs as well as a helper function `skylineInternal` (line 21 - 36) which contains an input parameter for defining whether the skyline results are distinct or not (line 22). The input variable `expr` contains a single tuple of type `(String, String)`. This is followed by `exprs` which contains a potentially empty sequence of tuples of the same schema with no size limit. The main purpose of `skyline` and `skylineDistinct` is to pass the input values to `skylineInternal` and set the input variable `distinct` accordingly (i.e. `true` for `skylineDistinct` and `false` otherwise). Both methods contain a `map()` function which seems to be redundant but using `exprs` directly causes the compiler to throw a type error (line 8 and line 17 - 18).

We note that using the combination of `expr` and `exprs` means that there must be **at least one** dimension in the skyline query. The result of a skyline query without any dimensions is equal to the input dataset. Requiring at least one dimension therefore only prevents redundant operators in a query and at the same time eliminates a potential source of errors introduced by the user.

In `skylineInternal`, we create a new `SkylineOperator` using a trivial factory stored as an object in `SkylineOperator` (line 26 - 35) that was omitted from the source code for brevity. This approach is comparable to writing a factory in a static method or using a similar singleton pattern. The value for `distinct` is passed as-is while the expressions from `expr` and `exprs` are mapped to `SkylineItemOption` using a factory (line 29 - 32). The first element in the tuple is first converted to a `Column` and subsequently `Expression` while the second parameter is converted to `SkylineMin`, `SkylineMax`, or `SkylineDiff`. The last necessary input value is the child which is a logical plan node. When using the DataFrame and DataSet APIs, the child and its descendants usually contain the logical plan generated by all previous chained methods. For example, the child plan of `df.filter([...]).skyline([...])` contains the plan generated by `df.filter([...])`.

Listing 5.16: Basic DataFrame and DataSet integration of skyline queries (Scala/Java) using string tuples

```
def skyline(
    expr: (String, String),
    exprs: (String, String)*
  ) : DataFrame =
    skylineInternal(
      distinct = false,
      expr,
      exprs.map(f => (f._1, f._2)): _*
    )

def skylineDistinct(
    expr: (String, String),
    exprs: (String, String)*
  ) : DataFrame =
    skylineInternal(
      distinct = true,
      (expr._1, expr._2),
      exprs.map(f => (f._1, f._2)): _*
    )

private def skylineInternal(
  distinct: Boolean,
  expr: (String, String),
  exprs: (String, String)*
): DataFrame = withPlan {
  SkylineOperator.createSkylineOperator(
    distinct,
    (expr +: exprs).map(
      f => SkylineDimension.createSkylineDimension(
        Column(f._1).expr,
        f._2
      )
    ),
    logicalPlan
  )
}
```

The second method is similar to the one using strings as input but accepts an arbitrary number of `Columns` instead. In this case, it is possible to leave the list of dimensions completely empty. The code can be found in Listing 5.17.

The expression `expr` of the `Column` (line 6) contains the necessary `SkylineDimensions` which means that it does not need to be created via a factory (line 7 - 8).

One difference is that the functions `skyline` (line 1 - 16) and `skylineDistinct` (line 18 - 20) are virtually duplicates instead of using a common function. This is due to some strange signature conflicts which arise when trying to pass a `Boolean` (for distinctiveness) and a list of `Columns` to the internal function. We omit the code for `skylineDistinct` here since it is exactly the same code as `skyline` except for setting `distinct = true` instead (line 19).

Lastly, we note that it is also possible to require at least one dimension using a method similar to Listing 5.16 We opt not to do this since python is then unable to find the method. The details of this particularity will be discussed in Section 5.5.

Listing 5.17: Basic DataFrame and DataSet integration of skyline queries (Scala/Java) using columns

```
1  def skyline(exprs: Column*): DataFrame = withPlan {
2    SkylineOperator.createSkylineOperator(
3      distinct = false,
4      (exprs).map(
5        f => {
6          f.expr match {
7            case SkylineDimension(child, minMaxDiff) =>
8              SkylineDimension(child, minMaxDiff)
9            case _ =>
10             throw new IllegalArgumentException([...])
11         }
12       }
13     ),
14     logicalPlan
15   )
16 }
17
18 def skylineDistinct(exprs: Column*): DataFrame = withPlan {
19   [...]
20 }
```

For the purpose of defining a skyline dimension on an existing `Column`, we introduce additional methods in Listing 5.18. These include `smin()`, `smax()`, and `sdiff()` which correspond to `SkylineMin`, `SkylineMax`, and `SkylineDiff` respectively. In each method, we create a new instance of `SkylineDimensions` stored as the expression of the column. This was already used in Listing 5.17.

Listing 5.18: Column methods for using columns in DataFrame/DataSet queries

```
1  def smin: Column = withExpr {
2    SkylineDimension.createSkylineDimension(expr, SkylineMin)
3  }
4
5  def smax: Column = withExpr {
6    SkylineDimension.createSkylineDimension(expr, SkylineMax)
7  }
8
9  def sdiff: Column = withExpr {
10   SkylineDimension.createSkylineDimension(expr, SkylineDiff)
11 }
```

To round off the integration, we also adapt the Domain Specific Language (DSL) provided by Spark SQL. For this, we introduce functions which in turn call the DataFrame/DataSet functions already introduced above. The source code can be found in Listing 5.19. For `skyline` and `skylineDistinct`, we create a new `SkylineOperator` where `distinct` is set correspondingly. We also use the `skylineDimensions` as well as the child logical plan (`logicalPlan`) as input parameters. The functions `smin`, `smax`, and `sdiff` create new `SkylineDimensions` according to their naming based on the input `Expression`.

Listing 5.19: Apache Spark Skyline Queries DSL

```
1  def skyline(skylineDimensions: SkylineDimension*)
2    : LogicalPlan = SkylineOperator(
3      SkylineIsNotDistinct, skylineDimensions, logicalPlan
4    )
5
6  def skylineDistinct(skylineDimensions: SkylineDimension*)
7    : LogicalPlan = SkylineOperator(
8      SkylineIsDistinct, skylineDimensions, logicalPlan
9    )
10
11 def smin(expr: Expression): SkylineDimension =
12   SkylineDimension(expr, SkylineMin)
13
14 def smax(expr: Expression): SkylineDimension =
15   SkylineDimension(expr, SkylineMax)
16
17 def sdiff(expr: Expression): SkylineDimension =
18   SkylineDimension(expr, SkylineDiff)
```

5.4 Simple Optimization

In this section, we introduce a single optimization to demonstrate the integration of optimization rules specific to skyline queries. Further optimizations exist and will be discussed in Section 5.8.

Given that skyline queries with no given dimension are possible (see Listing 5.17), a simple optimization is necessary. We use the fact that skyline queries with no dimensions have the original data set as result. The skyline operator can be removed from the query in its entirety.

We implement a new optimization rule for the Catalyst optimizer which can be found in Listing 5.20. Every optimization rule is of type `Rule[LogicalPlan]`, i.e., a rule for a logical plan node (line 3). Each rule has an `apply` function which takes a logical plan as input and outputs a modified logical plan (line 4 - 5). Optimizations can include adding, removing, and transforming nodes as well as replacing nodes in the tree completely.

We introduce a new non-instantiable non-extendable class `SkylineOptimizations` (line 1) which holds all optimizations for skyline operators. Every rule is a Scala `object` that extends `Rule[LogicalPlan]` and has a function `apply(plan:LogicalPlan): LogicalPlan` (line 3 - 4). We transform the plan using `transformUp` which transforms the plan in such a way that first all children (and descendants) are optimized before the node itself and its children are transformed (line 5). To achieve the actual transformation, we call a private function `removeEmptySkylines` which outputs a partial function (line 7 - 13). Partial functions in Scala are used when a function should only work on a "subset" of the input values. In our case, only a single function is necessary since it already handles all desired input cases. We use matching to find a `SkylineOperator` where the list of skyline dimensions is empty using `if skylineItems.isEmpty` (line 9 - 10). In this case, only the child logical node (and its descendants) are returned (line 11) while all other cases return the skyline operator as-is (line 12).

Listing 5.20: Spark SQL optimization rule for removing skyline operators without dimensions

```
final class SkylineOptimizations private { }

object RemoveEmptySkylines extends Rule[LogicalPlan] {
  override def apply(plan: LogicalPlan): LogicalPlan =
    plan transformUp removeEmptySkylines

  private val removeEmptySkylines
    : PartialFunction[LogicalPlan, LogicalPlan] = {
      case SkylineOperator(_, skylineDimensions, child)
        if skylineItems.isEmpty =>
          child
      case s @ SkylineOperator(_, _, _) => s
    }
}
```

To integrate the rule into Catalyst, we also need to add it to the `Optimizer` class. We add the rule to the default set of batches `defaultBatches` which contains a batch of rules that optimizes operators. A significantly shortened version of the implementation that contains all newly added parts can be found in Listing 5.21.

Listing 5.21: Adding the optimization rule from above to the set of optimization rules

```
def defaultBatches: Seq[Batch] = {
  val operatorOptimizationRuleSet =
    Seq(
        [...]
        RemoveEmptySkylines,
        [...]
      ) ++
      extendedOperatorOptimizationRules

  [...]
}
```

5.5 PySpark Integration

Given the minimum viable integration introduced in Section 5.3, it is already possible to formulate skyline queries using the Scala and Java DataFrame and DataSet APIs as well as query strings. Now, we look at the integration of skyline queries into PySpark. We note that formulating the skyline queries via query strings is already possible in PySpark given the changes introduced in Section 5.3. The additional changes in this section are required to make the skyline queries available in the DataFrame/DataSet API.

First, we need to introduce the `skyline` and `skylineDistinct` methods which work analogously to the corresponding Scala functions. The Scala functions are called internally using the remote execution framework Py4J [Py4]. The main difference is that the Python integration handles the text and columnar inputs in a single method while Scala provides separate methods.

The integration of the basic methods `skyline` (line 1 - 4) and `skylineDistinct` (line 6 - 9) can be found in Listing 5.22. They both work similarly by first calling the `_skyline` function that handles the input and returns a sequence (or rather `jseq`) of columns. The Py4J framework is subsequently used in the second line of the function to call `skyline` (line 3) or `skylineDistinct` (line 8) from the Scala integration. This always uses the method with column input from Scala since the inputs are always converted to columns in Python. The result of the call is turned into a DataFrame using the SQL context `self.sql_ctx` and returned (line 4 and line 9).

We now take a look at the internal `_skyline` function which can also be found in 5.22 on line 11 - 47.

If the list of inputs has only one element, we can convert it to the single element (line 14 - 15). We also convert the columns to Java columns by using the given `_to_java_column(c)` function (line 16). Next, we extract the value of `minMaxDiff` from the `kwargs` which contains the list of multiple dimensions. We use a default value of `"min"` for the argument (line 18 - 21).

We can convert the single argument to a Java column if only a single argument was passed via string (`isinstance(minMaxDiff, str)`) and the strings are non-empty (line 23 - 30). The functions `smin()`, `smax()`, and `sdiff()` (which will be discussed shortly) are called depending on which input was passed.

Lastly, we need to handle the case where a list of multiple dimensions was passed. We check for each item whether a value passed that conforms with the list of allowed values (`"min"`, `"max"`, and `"diff"`) and throw an error if one value does not (line 34 - 41). Otherwise, we use list comprehension to *zip* (i.e. generate pairs) the columns with the values for `minMaxDiff` and apply the corresponding function (`smin()`, `smax()`, and `sdiff()`) accordingly (line 42 - 45).

The result is either a single column or a list of columns. The result can be converted to a (Java) list that is subsequently used as input for the calls to the Scala methods (line 47).

Listing 5.22: `skyline` and `skylineDistinct` in PySpark

```
def skyline(self, *cols, **kwargs):
  skyline = self._skyline(cols, kwargs)
  jdf = self._jdf.skyline(skyline)
  return DataFrame(jdf, self.sql_ctx)

def skylineDistinct(self, *cols, **kwargs):
  skyline = self._skyline(cols, kwargs)
  jdf = self._jdf.skylineDistinct(skyline)
  return DataFrame(jdf, self.sql_ctx)

def _skyline(self, cols, kwargs):
  if not cols:
    raise ValueError("should be skyline by at least one column")
  if len(cols) == 1 and isinstance(cols[0], list):
    cols = cols[0]
  jcols = [_to_java_column(c) for c in cols]

  if (kwargs):
    minMaxDiff = kwargs.get('minMaxDiff', "min")
  else:
    minMaxDiff = ()

  if isinstance(minMaxDiff, str)
      and minMaxDiff and minMaxDiff.strip():
    if minMaxDiff.strip().lower() == "min":
      jcols = [ jc.smin() for jc in jcols ]
    elif minMaxDiff.strip().lower() == "max":
      jcols = [ jc.smax() for jc in jcols ]
    elif minMaxDiff.strip().lower() == "diff":
      jcols = [ jc.sdiff() for jc in jcols ]
    else:
      raise TypeError("only smin/smax/sdiff allowed for skyline," +
                      "but got %s" % str)
  elif isinstance(minMaxDiff, list):
    for item in minMaxDiff:
      if item.strip().lower() != "min"
        and item.strip().lower() != "max"
        and item.strip().lower() != "diff":
          raise TypeError(
            "only min/max/diff allowed for skyline," +
            "but got %s" % item )
    jcols = [ jc.smin() if mmd.strip().lower() == "min"
              else jc.smax() if mmd.strip().lower() == "max"
              else jc.sdiff()
              for mmd, jc in zip(minMaxDiff, jcols) ]

  return self._jseq(jcols)
```

All column functions are simple since we can use a predefined `_unary_op` method to invoke the correct operator. The implementation can be found in Listing 5.23. Note that `_smin_doc` is simply a string that holds the documentation (which is omitted here).

Listing 5.23: Columnar methods `smin`, `smax`, and `smin` in PySpark

```
1  smin = _unary_op("smin", _smin_doc)
2  smax = _unary_op("smax", _smax_doc)
3  sdiff = _unary_op("sdiff", _sdiff_doc)
```

We also define the functions `smin()`, `smax()`, and `sdiff()` independently of `Column` objects but with a parameter in Listing 5.24. These take either a column `col` or the name of the column as parameter. For example, `smin(col)` is equivalent to `col.smin()`. The `_invoke_function` method is used to invoke the correct function automatically.

Listing 5.24: Methods `smin`, `smax`, and `sdiff` in PySpark for skyline dimension generation

```
1  @since(3.1)
2  def smin(col):
3    return (
4      col.smin() if isinstance(col, Column)
5      else _invoke_function("smin", col)
6    )
7
8  @since(3.1)
9  def smax(col):
10   return (
11     col.smax() if isinstance(col, Column)
12     else _invoke_function("smax", col)
13   )
14
15 @since(3.1)
16 def sdiff(col):
17   return (
18     col.sdiff() if isinstance(col, Column)
19     else _invoke_function("sdiff", col)
20   )
```

5.6 Distributed Block-Nested-Loop

One way to optimize the skyline computation is to split the computation into local and global skylines. This can be done in Spark in a way similar to the algorithms used in Map-Reduce approaches (see Section 3.1). We base our implementation on the block-nested-loop algorithm introduced in Section 5.3.4.

The main idea is to have one execution node that computes the local skyline while a second node computes the global one. We note that there may be multiple instances of the local skyline node in this approach while there is only a single global skyline node which takes all local skylines as input. The input dataset is distributed using the Spark's internal clustering and partitioning for the local skylines.

We now look at the differences between the distributed skyline and the block-nested-loop algorithm found in Section 5.3.4. The algorithm of the distributed skyline nodes can be found in Listing 5.25 where we leave out unchanged blocks of code compared to Listing 5.13.

Since reusing the node for both local and global skyline computation is desirable, we introduce multiple distributions. This is done by modifying `requiredChildDistribution` (line 8 - 17). We offer three different distribution modes out of which only two are strictly necessary. The modes are selected depending on what is passed as the new parameter `requiredChildDistributionExpressions` (line 4) when the node is created:

- If `Some(expressions)` is passed where `expressions` is an empty sequence or list (line 10) then we do not distribute at all. We compute a global skyline like in the standalone block-nested-loop approach by using `AllTuples` as distribution (line 11) and force a single node with a single partition.
- If `Some(expressions)` is passed to the skyline operator and `expressions` is not empty (line 12), we create clusters according to `expressions` using `ClusteredDistribution(expressions)` (line 13). Tuples that have the same values for the expressions are clustered together. This case can potentially be used for optimizations using custom clusterings.
- If `None` is passed (line 14), we leave the distribution entirely up to Spark by using `UnspecifiedDistribution` (line 15).

Note that `Some()` and `None` correspond to the built-in `Option` idioms of Scala. As such, both of them can be used as parameter or return value of a function instead of using more primitive and error-prone `null` values.

Strictly speaking, we can choose to implement only the second or only the third option. Both make implementing local skylines possible.

The only change in `doExecute()` is that the window is from the function to within the `mapPartitionsInternal` mapping (line 30 - 31). As such, there is now a new window for each partition and the partitions can be computed independently from each other.

Listing 5.25: Block-nested-loop node fitting adapted for use in distributed skyline computations

```
1  case class BlockNestedLoopSkylineExec(
2    skylineDistinct: SkylineDistinct,
3    skylineDimensions: Seq[SkylineDimension],
4    requiredChildDistributionExpressions: Option[Seq[Expression]],
5    child: SparkPlan
6  ) extends BaseSkylineExec {
7
8    override def requiredChildDistribution: Seq[Distribution] = {
9      requiredChildDistributionExpressions match {
10       case Some(expressions) if expression.isEmpty =>
11         AllTuples :: Nil
12       case Some(expressions) =>
13         ClusteredDistribution(expressions) :: Nil
14       case None =>
15         UnspecifiedDistribution :: Nil
16     }
17   }
18
19   override protected def doExecute(): RDD[InternalRow] = {
20     val childOutputSchema = child.output.map { f => f.dataType }
21     val skylineDimensionOrdinals = skylineDimensions.map { option =>
22       child.output.map{ attr => attr.exprId }
23         .indexOf(option.child.references.head.exprId)
24     }
25     child.execute().mapPartitionsInternal { partitionIter =>
26       val blockNestedLoopWindow =
27         new mutable.ArrayBuffer[InternalRow]()
28       partitionIter.foreach { row =>
29         var isDominated = false; var breakWindowCheck = false
30         val dominatedSkylineTuples =
31           new mutable.ArrayBuffer[InternalRow]()
32         val iter = blockNestedLoopWindow.iterator
33         while (iter.hasNext && !breakWindowCheck) {
34           val windowRow = iter.next()
35           val dominationResult = [...]
36           dominationResult match { [...] }
37         }
38         if (!isDominated) {
39           if (dominatedSkylineTuples.nonEmpty) {
40             blockNestedLoopWindow --= dominatedSkylineTuples }
41           blockNestedLoopWindow.append(row.copy())
42         }
43       }
44       numOutputRows += blockNestedLoopWindow.size
45       blockNestedLoopWindow.toIterator
46     }
47   }
48 }
```

Introducing distinct local and global nodes makes the planning more complex. The modified strategy can be found in Listing 5.26. We change the `Strategy` for planning the physical skyline nodes in Listing 5.26 such that we now delegate the decisions to a new utility (line 10 - 14).

Listing 5.26: Spark strategy which calls a utility to plan the skyline

```
1  abstract class SparkStrategies extends QueryPlanner[SparkPlan] {
2    [...]
3
4    object BasicOperators extends Strategy {
5      def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
6        [...]
7        case SkylineOperator(
8          distinct, skylineDimensions, child
9        ) =>
10         SkylineUtils.planSkyline(
11           distinct,
12           skylineDimensions,
13           planLater(child)
14         )
15       case _ => Nil
16     }
17   }
18 }
```

The newly introduced utility `SkylineUtils` can be found in Listing 5.27 and is divided into two methods.

First, we take a look at the helper method `createSkyline` (line 2 - 18) that is used to create new skyline nodes for the physical plan. The input parameters `skylineDistinct` and `skylineDimensions` (line 3 and 4) correspond to the values retrieved from a logical plan node. To enable distribution of the nodes and distinguishing between local and global skyline computation, `requiredChildDistributionExpressions` (line 5) defines a sequence of expression which specifies which distribution is used. We also use a flag `globalSkyline` (line 6) to distinguish between creating local and global skyline execution nodes. The parameter `child` is defines the child of the node.

In case of global skylines (line 8 - 11), we create a new node with the passed information where the distribution expressions are set to `Some(Nil)`. This forces a global skyline distribution where all data is sent to a single node using the `AllTuples` distribution (see Listing 5.25). Otherwise, in case of local skylines (line 12 - 17), we pass an expression on which the distribution is based. If `None` is passed instead (as is here the case), Spark automatically chooses a distribution if it is left unspecified. This is our default behavior for complete local skylines.

The main method `planSkyline` (line 20 - 31) takes the information retrieved from the logical plan node as parameters (line 21 - 23). From this, a local skyline is constructed (line 24 - 26) using `child` as "input" node (usually not yet planned due to `planLater()` – also see Listing 5.26). We set `globalSkyline` to `false` (line 26). The local skyline serves as `child` to the global skyline(line 27 - 29). All parameters, except for `globalSkyline` which is set to `true` (line 29), and the child (line 29) are the same as in the local skyline when creating the global skyline node.

Listing 5.27: Planning utility for planning distributed block-nested-loop skyline algorithms

```
1  object SkylineUtils {
2    private def createSkyline(
3       skylineDistinct: SkylineDistinct,
4       skylineDimensions: Seq[SkylineDimension],
5       requiredChildDistributionExpressions: Option[Seq[Expression]],
6       globalSkyline: Boolean,
7       child: SparkPlan ): SparkPlan = {
8     if (globalSkyline) {
9        BlockNestedLoopSkylineExec(
10         skylineDistinct, skylineDimensions, Some(Nil),
11         child )
12    } else {
13      BlockNestedLoopSkylineExec(
14        skylineDistinct, skylineDimensions,
15        requiredChildDistributionExpressions,
16        child )
17    }
18   }
19
20   def planSkyline(
21      skylineDistinct: SkylineDistinct,
22      skylineDimensions: Seq[SkylineDimension],
23      child: SparkPlan ) : Seq[SparkPlan] = {
24     val localSkyline: SparkPlan = createSkyline(
25       skylineDistinct, skylineDimensions, None,
26       globalSkyline = false, child )
27     val globalSkyline = createSkyline(
28       skylineDistinct, skylineDimensions, Some(Nil),
29       globalSkyline = true, localSkyline )
30     globalSkyline :: Nil
31   }
32 }
```

5.7 Incomplete Skylines Based on Distributed Block-Nested-Loop

We can adapt the (distributed) block-nested-loop algorithm to deal with incomplete datasets. This skyline algorithm will be used automatically if at least one of the skyline dimensions is `nullable`. At the same time, we also provide an override that forces Spark to use the complete algorithm. We introduce this because the complete algorithm is faster than the incomplete one if no `null` values are encountered. In this case, both algorithms also yield the same result.

To implement the override in Spark SQL query strings, we extend the skyline query string syntax from [BKS01] by introducing the `COMPLETE` keyword for the skyline clause. The modified query string syntax can be found in Listing 5.28.

Listing 5.28: Syntax of skyline queries in SQL based on [BKS01] including override for complete skylines

```
1 SKYLINE OF [ DISTINCT ] [ COMPLETE ]
2 d_1 [MIN | MAX | DIFF], ..., d_m [MIN | MAX | DIFF]
```

Analogously, we also define two new methods for the DataFrame and DataSet APIs:

- `skylineComplete`
- `skylineDistinctComplete`

We omit the details about the implementation of these small changes since they are simple changes to the minimally viable integration (see Section 5.3).

The `DominanceUtils` must be adapted and can be found in Listing 5.29 which is virtually identical to Listing 5.14 save for the following differences:

- We introduce a new boolean variable `skipNullValues` that indicates whether to disregard the comparison if one of the two rows is `null` for a given dimension (line 8).
- We introduce a new boolean variable `skipMismatchingNulls` (line 9). It allows us to skip pairs of incomparable (according to the definition of incomparability in incomplete skylines) tuples entirely (line 15 - 20).
- We introduce a check that determines whether one of the rows is `null` for the given dimension (line 25 - 26). If this is the case and `skipNullValues` is `true`, then we skip the dimension. Otherwise the compare as before (shortened in line 28 - 32).

The rest of the dominance utilities remains as-is, and the behavior of the utility is unchanged if `skipNullValues` is set to `false`.

Listing 5.29: Utilities for checking dominance for complete or incomplete two tuples `rowA` and `rowB`

```
1  sealed class DominanceUtils private() {}
2    def checkRowDominance(
3      rowA: InternalRow,
4      rowB: InternalRow,
5      schema: Seq[AbstractDataType],
6      ordinals: Seq[Int],
7      minMaxDiff: Seq[SkylineMinMaxDiff],
8      skipNullValues: Boolean,
9      skipMismatchingNulls: Boolean = false
10   ): DominanceResult = {
11     var a_strictly_better = false;
12     var b_strictly_better = false;
13     var a_different_b = false;
14
15     if (skipMismatchingNulls) {
16       if (ordinals.exists ( ord =>
17           rowA.isNullAt(ord) != rowB.isNullAt(ord) ) ) {
18         return Incomparability
19       }
20     }
21
22     ordinals.zipWithIndex.foreach { f =>
23       val (ord, idx) = (f._1, f._2)
24
25       if ( skipNullValues &&
26            ( rowA.isNullAt(ord) || rowB.isNullAt(ord) ) ) {
27         // skip dimension where at least one is null
28       } else {
29         schema(ord) match {
30           case ByteType | ShortType | IntegerType =>
31                   [...]
32         }
33       }
34     }
35
36     if (a_different_b) {
37       Incomparability
38     } else if (a_strictly_better && !b_strictly_better) {
39       Domination
40     } else if (b_strictly_better && !a_strictly_better) {
41       AntiDomination
42     } else if (a_strictly_better && b_strictly_better) {
43       Incomparability
44     } else {
45       Equality
46 } } }
```

Clustering by `null` values can be achieved by passing the Spark expression `IsNotNull()` or `IsNull()`. We will discuss this the distribution using these expressions later in this chapter.

For each cluster, the transitivity property holds. There exists no scenario in which a tuple q which is already dominated in the local skyline by a tuple a can dominate another tuple from a different cluster that isn't also dominated by a. The only difference to the "regular" distributed local skyline is that the columns which contain `null` values must be skipped. A specialized skyline node is therefore only needed for the global skyline. This node must deal with the particularities of cyclic dominance and the loss of transitivity property.

The modified algorithm based on the distributed block-nested-loop variant from Listing 5.25 can be found in Listing 5.30. We add the parameter `isIncompleteSkyline` (line 5) which is passed to the dominance utilities to skip `null` values for each column that contains only `null`s according to the clustering through `skipNullValues` (line 31). We also tell the node to set `skipMismatchingNulls` (line 32) such that we never under any circumstances compare tuples from different clusters, i.e., with `null` values in different columns. This is necessary since spark still may assign tuple with different `null` values to the same partition.

Listing 5.30: Block-nested-loop node adapted for local skyline computations on incomplete data

```
1  case class BlockNestedLoopSkylineExec(
2    skylineDistinct: SkylineDistinct,
3    skylineDimensions: Seq[SkylineDimension],
4    requiredChildDistributionExpressions: Option[Seq[Expression]],
5    isIncompleteSkyline: Boolean,
6    child: SparkPlan
7  ) extends BaseSkylineExec {
8
9    override def requiredChildDistribution: Seq[Distribution] = [...]
10
11   override protected def doExecute(): RDD[InternalRow] = {
12     val childOutputSchema = child.output.map { f => f.dataType }
13     val skylineDimensionOrdinals = skylineDimensions.map { option =>
14       child.output.map{ attr => attr.exprId }
15         .indexOf(option.child.references.head.exprId)
16     }
17     child.execute().mapPartitionsInternal { partitionIter =>
18     val blockNestedLoopWindow =
19         new mutable.ArrayBuffer[InternalRow]()
20       partitionIter.foreach { row =>
21         var isDominated = false; var breakWindowCheck = false
22         val dominatedSkylineTuples =
23           new mutable.ArrayBuffer[InternalRow]()
24         val iter = blockNestedLoopWindow.iterator
25         while (iter.hasNext && !breakWindowCheck) {
26           val windowRow = iter.next()
27           val dominationResult = DominanceUtils.checkRowDominance(
28             row, windowRow,
29             childOutputSchema, skylineDimensionOrdinals,
30             skylineDimensions.map { f => f.minMaxDiff },
31             skipNullValues = isIncompleteSkyline,
32             skipMismatchingNulls = isIncompleteSkyline
33           )
34           dominationResult match { [...] }
35         }
36         if (!isDominated) {
37           if (dominatedSkylineTuples.nonEmpty) {
38             blockNestedLoopWindow --= dominatedSkylineTuples }
39           blockNestedLoopWindow.append(row.copy())
40         }
41       }
42       numOutputRows += blockNestedLoopWindow.size
43       blockNestedLoopWindow.toIterator
44     }
45   }
46 }
```

The code for the global skyline node can be found in Listing 5.31. It is similar to Listing 5.25 which contains the already introduced version for complete datasets.

One of the main differences of the incomplete version compared to the complete version is that the child distribution is always set to `AllTuples` since it always computes the global skyline (line 6 - 7).

We also introduce a new `datasetWindow` (line 18) which is filled with the full dataset of the partition (line 21 - 23). This is necessary since we need to compare every tuple in the window with every other remaining tuple. Therefore, we iterate over the `datasetWindow` instead of the `resultWindow` which is now reduced to holding the result of the computation. Since we have an incomplete dataset, we need to skip missing values in the dominance computation which is achieved by setting `skipNullValues` in line 35. There is now no scenario in which we exit the inner loop prematurely. The flag to achieve this is therefore removed.

We add a tuple to the `resultWindow` if and only if it was not dominated by any other tuple (line 46). The result of the computation is, as mentioned above, the content of `resultWindow` of which we return an iterator (line 48) for each iteration of `mapPartitionsInternal` (line 17).

Listing 5.31: Physical plan node for (global) incomplete skyline queries

```
case class IncompleteSkylineExec(
  skylineDistinct: SkylineDistinct,
  skylineDimensions: Seq[SkylineDimension],
  child: SparkPlan
) extends BaseSkylineExec {
  override def requiredChildDistribution: Seq[Distribution] =
    AllTuples :: Nil

  override protected def doExecute(): RDD[InternalRow] = {
    val childOutputSchema = child.output.map { f => f.dataType }
    val skylineDimensionOrdinals =
      skylineDimensions.map { option =>
        child.output.map{ attr => attr.exprId
      }.indexOf(option.child.references.head.exprId)
    }

    child.execute().mapPartitionsInternal { partitionIter =>
      val datasetWindow = new mutable.HashSet[InternalRow]()
      val resultWindow = new mutable.HashSet[InternalRow]()

      partitionIter.foreach { row =>
        datasetWindow += row.copy()
      }

      datasetWindow.foreach { row =>
        var isDominated = false;

        datasetWindow.foreach { windowRow =>
          val dominationResult = DominanceUtils.checkRowDominance(
            row,
            windowRow,
            childOutputSchema,
            skylineDimensionOrdinals,
            skylineDimensions.map { f => f.minMaxDiff },
            skipNullValues = true
          )

          dominationResult match {
            case Domination =>
            case AntiDomination =>
              isDominated = true
            case Equality | Incomparability | _ => // NO ACTION
          }
        }

        if (!isDominated) { resultWindow += row }
      }
      resultWindow.toIterator
} } }
```

We also expand `planSykline` in the planning utilities which can be found in Listing 5.32. To enable the forced completeness override, we introduce a new parameter `skylineComplete` (line 3). Using this, we can compute whether to use a complete or incomplete algorithm by distinguishing three cases:

- If the override **is** set, then we do **not** use an incomplete algorithm and therefore set `dimensionNullable` to `false` (line 18 and 19).
- If the override is **not** set, we check for each dimension whether it is `nullable` (line 18 and 20). We use the predefined `exists` function to check whether at least one dimension exists which is `nullable`. This yields two different sub-cases for automatic computation:
  - If at least one dimension is `nullable`, then we set `dimensionNullable` to `true` and use an incomplete algorithm (line 18 and 20).
  - Otherwise, if all dimensions are **not** `nullable`, then we can use a complete algorithm by setting `dimensionNullable` to `false` (line 18 and 20).

When creating local and global skyline, we now also pass the `dimensionNullable` variable to the `createSkyline` function to distinguish between complete and incomplete algorithms being created (line 28 and 37).

Additionally, we also introduce a `SkylineForceBNL` option which can be passed as an alternative to `SkylineIsComplete`. In the query string, we can use `BNL` instead of `COMPLETE`. We do not show the integration of this additional option as it is only intended for testing and benchmarking and only requires trivial additions beyond the existing changes. If `SkylineForceBNL` was passed (line 8), we return a plan for a block-nested-loop skyline immediately (line 9 - 15).

Listing 5.32: Planning utility decision between complete and incomplete

```
def planSkyline(
     skylineDistinct: SkylineDistinct,
     skylineComplete: SkylineComplete,
     skylineDimensions: Seq[SkylineDimension],
     child: SparkPlan
   ) : Seq[SparkPlan] = {

    if (skylineComplete == SkylineForceBNL) {
      return BlockNestedLoopSkylineExec(
        skylineDistinct,
        skylineDimensions,
        Some(Nil),
        isIncompleteSkyline = false,
        child
      ) :: Nil
    }

    val dimensionNullable = skylineComplete match {
      case SkylineIsComplete => false
      case _ => skylineDimensions.exists(_.child.nullable)
    }

    val localSkyline: SparkPlan = createSkyline(
      skylineDistinct,
      skylineDimensions,
      None,
      globalSkyline = false,
      dimensionNullable,
      child
    )

    val globalSkyline = createSkyline(
      skylineDistinct,
      skylineDimensions,
      Some(Nil),
      globalSkyline = true,
      dimensionNullable,
      localSkyline
    )

    globalSkyline :: Nil
  }
```

We also introduce new cases for incomplete skylines in the function `createSkyline()`. The additions can be found in Listing 5.33:

- If a global skyline is created (`globalSkyline` is `true`) for an incomplete dataset (`incompleteSkyline` is `true`) then a new `IncompleteSkylineExec` (Listing 5.31) is created (line 1 - 6).
- If we create a local skyline for an incomplete dataset (`incompleteSkyline` is `true`), then we reuse the regular block-nested-loop algorithm node from Listing 5.25 (line 7 - 15). For each skyline dimension, we create a new expression `IsNull(dimension)` which is used for clustering by passing them as the required child distribution (line 11). This causes tuples with the same missing values to end up in the same partition. Additionally, we also need to skip the missing values in the dominance checks which we enable by setting `isIncompleteSkyline` to `true` (line 12).

Listing 5.33: Creating local and global computation nodes for skylines on incomplete datasets

```
1  if (incompleteSkyline && globalSkyline) {
2    IncompleteSkylineExec(
3      skylineDistinct,
4      skylineDimensions,
5      child
6    )
7  } else if (incompleteSkyline) {
8    BlockNestedLoopSkylineExec(
9      skylineDistinct,
10     skylineDimensions,
11     Some(skylineDimensions.map { f => IsNull(f.child) }),
12     isIncompleteSkyline = true,
13     child
14   )
15 }
```

5.8 Advanced Catalyst Optimizations

In this section, we introduce further optimizations besides the one already discussed in Section 5.4. Since all optimizations follow the same schema, we leave out the basic explanations and focus on the concrete optimizations and particularities here.

5.8.1 Removing Redundant Skyline Dimensions

One of the simplest optimizations is to remove redundant skyline dimensions since the duplicates do not change the result of the skyline computation. A corresponding optimization rule which, like all optimizations, extends the predefined `Rule` can be found in Listing 5.34.

The solution is to simply match all non-empty skyline operators (line 7) and then call the predefined Scala function `distinct()` on the dimensions (line 10). This will remove any duplicate skyline dimensions.

Listing 5.34: Optimization to remove redundant dimensions from a skyline operator

```
1  object RemoveRedundantSkylineDimensions extends Rule[LogicalPlan] {
2    override def apply(plan: LogicalPlan): LogicalPlan =
3      plan transform removeRedundantDimensions
4
5    private val removeRedundantDimensions:
6    PartialFunction[LogicalPlan, LogicalPlan] = {
7      case SkylineOperator(distinct, complete, skylineItems, child)
8      if skylineItems.nonEmpty =>
9        SkylineOperator(
10         distinct, complete, skylineItems.distinct, child
11       )
12     case s @ SkylineOperator(_, _, _, _) => s
13   }
14 }
```

5.8.2 Pushing Skylines Through Joins

In this section, we implement a simple optimization which allows us to push non-distinct skylines through joins (based on Section 2.5.1). The main advantage of this is that the skyline can then be computed earlier. By doing this, the input sets for both the skyline and the join decrease in size which potentially makes the computation faster.

The code for this optimization can be found in Listing 5.35. Like every other optimization, this one also extends `Rule[LogicalPlan]` (line 1) and therefore provides an `apply` method (line 2 - 3). We take an existing plan and transform it if and only if the non-distinct `SkylineOperator` contains a `Join` with a `Project` in between (line 7 - 8). This `Project` in between is inserted by Spark in an earlier translation or optimization step and is present in all cases encountered during testing this optimization. We distinguish between three different cases for applying this optimization rule:

- In the first case, we check whether all skyline dimensions can be found on the "left" side of the join. This is done by first getting the expression ids `exprId` (line 11) for each dimension (line 10) and then subsequently checking whether all of those expression ids (`forall`) can be found in the expression ids of the output of the left side (line 12). Similarly, we also ensure that none of the dimensions appear (`exists`) on the right side of the join (line 13 - 15). If all those conditions are met, we (re-)construct the query by putting the skyline on the left side of the join (line 17).
- The checks in the second case (line 20 - 25) are similar to the first case but with the left and right side switched. If the conditions are met, then we (re-)construct the join analogously with the difference that we now replace the right side of the join (line 27).
- We return the original skyline if no matching case (i.e., no skyline over a join) was identified (line 29).

Note that these conditions make the push-through rules applicable to all types of join including the outer joins. A more detailed push-through rule could be used to handle non-outer joins where the skyline dimensions also include one or more "join columns". This case cannot as easily be handled for the outer joins.

Listing 5.35: Optimization for pushing skyline operators into joins if they only need to be applied to one "side" of the join

```
1  object PushSkylineThroughJoin extends Rule[LogicalPlan] {
2    override def apply(plan: LogicalPlan): LogicalPlan =
3      plan transform pushSkylineThroughJoin
4
5    private val pushSkylineThroughJoin
6        : PartialFunction[LogicalPlan, LogicalPlan] = {
7      case s @ SkylineOperator(SkylineIsNotDistinct, _, dimensions,
8        p @ Project(_, j @ Join(left, right, _, _, _) ) ) =>
9          if (
10           dimensions.map(_.child.references.head)
11               .map(_.exprId)
12               .forall(left.output.map(_.exprId).contains)
13           && !dimensions.map(_.child.references.head)
14               .map(_.exprId)
15               .exists(right.output.map(_.exprId).contains)
16         ) {
17           p.copy(child = j.copy(left = s.copy(child = left)))
18         }
19         else if (
20           dimensions.map(_.child.references.head)
21               .map(_.exprId)
22               .forall(right.output.map(_.exprId).contains)
23           && !dimensions.map(_.child.references.head)
24               .map(_.exprId)
25               .exists(left.output.map(_.exprId).contains)
26         ) {
27           p.copy(child = j.copy(right = s.copy(child = right)))
28         }
29         else { s }
30   }
31 }
```

5.8.3 Rewriting Skyline Queries with Only One Dimension

Skyline queries with only a single dimension can be rewritten as efficient regular queries that do not require full skyline computations and can benefit from other optimizations of Spark. The rule for rewriting such queries is more complex than any other rule introduced up to this point and can be found in Listing 5.36.

In the first set of transformations of the logical plan, we only look at skyline operators with a single dimension (line 9) which is either minimized (line 10) or maximized (line 12). The main idea is to find the minimum (for minimization) or maximum (for maximization) of the skyline dimension and then filter the dataset such that only tuples remain which have that value. If the skyline is distinct, then we simply select the topmost result.

To achieve this, we first get the skyline dimension (line 13) and create a projection node that projects all tuples to only the skyline dimension (line 14 - 15). Using an aggregate (line 16 - 27) we can then create a node that calculates the minimum (line 22 - 24) or maximum (line 19 - 21) without any grouping (line 17). The projection created above serves as an input for the aggregate (line 26).

Since the result of this aggregate is always a single value, we can write it as a scalar subquery which is a subquery that only returns a single value and is heavily optimized in Spark (line 28).

Using a filter, we can then select the tuples in `child` which are equal to the value returned by the subquery for the given skyline dimension (line 29 - 34).

If the skyline is distinct (line 36), we add a `Limit` of 1 to the query (line 37) which has the filter as its input. The local and global limits are necessary since some optimizations in Spark are only performed if both of them are present. In case the skyline is not distinct, we simply return the filter itself which then may yield multiple values (line 38).

If a non-distinct skyline has only a single `DIFF` dimension, then we can simply remove the skyline operator since no dominance is possible according to the definition (see Section 2.3.1). We match these cases (line 44 - 47) and simply return the child (line 48).

A distinct skyline operator with exactly a single DIFF skyline dimension (line 39 - 42) can be replaced by `Deduplicate` to remove duplicates for the given skyline dimension (line 43).

Listing 5.36: Optimization for rewriting one-dimensional skylines

```
object RemoveSingleDimensionalSkylines extends Rule[LogicalPlan] {
  override def apply(plan: LogicalPlan): LogicalPlan =
    plan transform removeSingleDimensionalSkylines

  private val removeSingleDimensionalSkylines:
  PartialFunction[LogicalPlan, LogicalPlan] = {
    case SkylineOperator(distinct, _, skylineItems, child)
      if (
        skylineItems.size == 1
        && ( skylineItems.head.minMaxDiff == SkylineMin
             || skylineItems.head.minMaxDiff == SkylineMax )
      ) =>
        val skylineDimension = skylineItems.head
        val projection = Project( skylineDimension.child
          .asInstanceOf[NamedExpression] :: Nil, child )
        val aggregate = Aggregate(
          Nil,
          skylineItems.head.minMaxDiff match {
            case SkylineMax => Alias(
              Max(skylineDimension.child).toAggregateExpression(),
              "max(" + skylineDimension.child.sql + ")")() :: Nil
            case _ => Alias(
              Min(skylineDimension.child).toAggregateExpression(),
              "min(" + skylineDimension.child.sql + ")")() :: Nil
          },
          projection
        )
        val subQuery = ScalarSubquery(aggregate)
        val filter = Filter(
          And(
            IsNotNull(skylineDimension.child),
            EqualTo(skylineDimension.child, subQuery)
          ), child
        )

      if (distinct == SkylineIsDistinct) {
        GlobalLimit(Literal(1), LocalLimit(Literal(1), filter))
      } else { filter }
    case SkylineOperator(
        _@SkylineIsDistinct, _, skylineItems, child
      ) if skylineItems.size == 1
          && skylineItems.head.minMaxDiff == SkylineDiff =>
        Deduplicate(skylineItems.head.child.references.toSeq, child)
    case SkylineOperator(
        _@SkylineIsNotDistinct, _, skylineItems, child
      ) if skylineItems.size == 1
          && skylineItems.head.minMaxDiff == SkylineDiff =>
        child
} }
```

We note that the node `Deduplicate` in the logical plan cannot be directly translated to a physical plan and must be replaced by an aggregate. This is achieved by the (existing) `ReplaceDeduplicateWithAggregate` optimization rule. Given that we now have multiple skyline optimizations, we move them to a new `Batch` (Listing 5.37) within the set of optimization rules. We ensure that `ReplaceDeduplicateWithAggregate` comes after `RemoveSingleDimensionalSkylines` (line 4 - 5). Additionally, we ensure that the skylines are pushed through joins if they can be done one a single "side" of the join (line 6).

Listing 5.37: Batch of skyline query optimizations

```
1  Batch("Skyline", fixedPoint,
2        RemoveEmptySkylines,
3        RemoveRedundantSkylineDimensions,
4        RemoveSingleDimensionalSkylines,
5        ReplaceDeduplicateWithAggregate,
6        PushSkylineThroughJoin)
```

5.9 Limitations

In this section, we discuss the limitations encountered during the implementation phase. For non-implemented algorithms we also give the reason(s) for not implementing them as a part of this thesis. The section is structured by algorithms and features which represent a subsection each.

5.9.1 Divide-and-Conquer Algorithms

The divide-and-conquer algorithm (Section 2.4.4) divides the dataset in to multiple parts which are then processed individually before generating a full skyline from the parts.

This is roughly equivalent to our distributed implementation sans parallelism. As such, this algorithm was dropped in favor of the block-nested-loop algorithm since most of its advantages are already achieved by distribution. We can even consider the distributed algorithm to be a specialized version of the divide-and-conquer approach.

5.9.2 Index-Based Algorithms

We did not implement any index-based algorithms (Section 2.4.5) as Spark SQL does not support indexes. As such, the only possibilities are to either massively extend Spark to add support for indexes or to build the indexes during the skyline computation.

The former is well outside of the scope of this thesis as it would require tremendous effort. Doing the latter adds a costly step of computing the index which can then subsequently be used to generate the skyline. Since the potential performance improvements of this

are questionable, we did not implement either approach as both would push the thesis in scope.

Another possibility would be to save the index in SparkSQL. This brings a plethora of problems that include but are not limited to keeping the index updated and saving the index. It follows that doing so is also vastly outside of the scope of this thesis.

5.9.3 Pushing Skylines into Joins

Pushing skylines into joins (Section 2.5.2) is an optimization technique where skylines are computed before the join and then once again after. This yields potentially better performance in some special cases. In Spark SQL, this would require a re-implementation or tricky use of existing joins. The potential performance improvements do not justify the effort to implement this optimization as a part of this thesis.

5.9.4 Advanced Partitioning Schemes

We leave the partitioning schemes (Section 3.1.3) largely to Spark since the predefined partitioning already does a reasonable job of partitioning the data properly. It also takes the prior existing partitioning into account which may lead to locality helping with the performance of the query. Other partitioning schemes may be achieved by modifying the `Expression` which governs the partitioning.

5.9.5 Algorithms using Advanced Partitioning Schemes

The more advanced algorithms introduced for MapReduce frameworks (Section 3.1.4 and Section 3.1.5) were not fully implemented due to the following reasons:

- Passing meta-data between the nodes of Spark is hard to achieve. This makes computing the best and worst point of a grid harder since there is no easy way to pass on the best and worst points of the relevant (dominating) grid cells.
- Spark only guarantees that all points from the same grid/group end up in the same partition. We would have to ensure manually that each partition only considers a single grid or group by performing a manual separation of the input data.
- Calculating the Z-addresses of the data points in Spark is computationally expensive as they need to be recomputed each time. Also, having to regard the different data types also adds complexity. This approach is therefore also outside of the reasonable scope of this thesis due to complications which are introduced.

However, the implementation of the skyline algorithm in Spark still draws from some ideas and plots which were presented in the papers that introduce the mentioned approaches.

5.10 Summary

In this chapter, we have taken a look at how skyline queries can be integrated into Spark. We have covered both the basic integration as well as more advanced techniques.

First, in Section 5.1, we have looked at how the results of the integration can be used and built a basis for understanding skyline queries in Spark. Based on this "how to use" section, we have subsequently looked at the impacts of source code modification versus using the experimental extension API. We have concluded that the source code modification is viable and its advantages far outweigh its disadvantages when compared to the extension API.

Based on this decision, we have then shown how a minimum viable integration of skyline queries into Spark can be done in Section 5.3. The basic aspects have included parsing query strings, changes to the analyzer, planning, basic optimizations, and a block-nested-loop algorithm. This minimally viable integration is fully functional and can serve as a basis for more advanced implementations and optimizations. We have also integrated the skyline queries into the PySpark API in Section 5.5. This allows the skyline API to be used in Python programs and builds on our previous implementations.

To extend the block-nested-loop algorithm and improve performance, we have then also implemented a distributed algorithm which uses multiple nodes. This can be found in Section 5.6. It was based largely on the MapReduce approaches to skyline queries as the "phases" from the MapReduce can be emulated by Spark with relative ease by using multiple nodes. For distribution, we have used the pre-defined distribution mechanisms of Spark SQL to make local and global skyline computations possible.

As an extension to the functionality of skyline queries, we have also implemented the computation of skylines on incomplete data in Section 5.7. Here, special care must be taken to distribute the data in clusters and make sure that only data from the same cluster is processed together.

Lastly, we have also implemented some more advanced optimizations which were not included in the basic optimizations. The following optimizations can be found in Sections 5.4 and 5.8:

- Removing skylines without dimensions
- Removing redundant skyline dimensions
- Pushing skylines through joins
- Rewriting single-dimensional skyline queries to "plain" SQL using aggregates

To round the chapter off, we have taken a look at limitations which were encountered during the implementation in Section 5.9.

CHAPTER 6

Testing and Performance Evaluation

In this chapter, we will first take a look at the testing of our implementation of skyline queries in Spark including the source code used for testing. Then, we will look at the performance of the implementation via benchmarks. For this, we will describe the benchmarks themselves before we get to the obtained results. All plots visualizing the performance in this chapter were drawn using the `ggplot2` package of the R programming language.

6.1 Unit Testing

In this section, we will take a look at the unit tests intended to verify the functionality of the skyline both step-by-step and through the entire Spark technology stack. All tests are included in existing (unit) `test` folders of Spark and can be executed either manually or via the build tools.

We note that most files intended for testing contain long sections of similar code. Therefore, we will partially omit the descriptions and code of test cases if they are similar to other cases which were already discussed.

Some code fragments in this section are split to accommodate code which does not fit reasonably on a single page. For each split code fragment, we will indicate the part number in the caption of the corresponding listing.

6.1.1 Test Query String Parsing

First, we test the parser responsible for translating query strings to (unresolved) logical plans. The code for these tests can be found in Listing 6.1. We compare the logical plan generated by the parser to the expected plan which is given using the Spark SQL DSL language. All tests belong to a new `test` method (line 1). We use `asserEqual` to check whether the generated plan matches the expected plan. Using the DSL instead of using logical skyline operator nodes directly may potentially introduce an additional source of errors but makes the tests much more concise and readable.

The parser tests include the following test cases:

- Parsing a simple skyline query string where only a single dimension is minimized (line 2 - 5)
- Parsing a skyline query string with three dimensions where each type of skyline (`MIN`, `MAX`, and `DIFF`) is present (line 6 - 13)
- Parsing a skyline query strings which includes `DISTINCT` with three dimensions (line 14 - 21)
- Parsing a skyline query string which includes `COMPLETE` with three dimensions (line 22 - 29)
- Parsing a skyline query string which contains both `DISTINCT` and `COMPLETE` with three dimensions (line 30 - 38)

We note that the parsing of skyline strings in case-insensitive in Spark. Any capitalized keywords only increase readability and do not affect the parsing of the query.

Listing 6.1: Test parsing of skyline query strings

```
1  test("skyline") {
2    assertEqual(
3      "select * from t skyline of a min",
4      table("t").select(star()).skyline(table("t").smin('a))
5    )
6    assertEqual(
7      "select * from t skyline of a min, b max, c diff",
8      table("t").select(star()).skyline(
9        table("t").smin('a),
10       table("t").smax('b),
11       table("t").sdiff('c),
12     )
13   )
14   assertEqual(
15     "select * from t skyline of distinct a min, b max, c diff",
16     table("t").select(star()).skylineDistinct(
17       table("t").smin('a),
18       table("t").smax('b),
19       table("t").sdiff('c),
20     )
21   )
22   assertEqual(
23     "select * from t skyline of complete a min, b max, c diff",
24     table("t").select(star()).skylineComplete(
25       table("t").smin('a),
26       table("t").smax('b),
27       table("t").sdiff('c),
28     )
29   )
30   assertEqual(
31     "select * from t skyline of distinct complete " +
32     "a min, b max, c diff",
33     table("t").select(star()).skylineDistinctComplete(
34       table("t").smin('a),
35       table("t").smax('b),
36       table("t").sdiff('c),
37     )
38   )
39 }
```

6.1.2 Test Removing Redundant Skyline Dimensions

Next, we test whether the redundant skyline dimensions are removed from the query plan correctly if applicable. The code for testing the removal of redundant skyline dimensions can be found in Listing 6.2. This also includes a test for the special case where the skyline dimensions are empty and the skyline can be omitted.

For these tests, we analyze the original query as well as the query with applied optimizations. We do not use the full optimizer with all optimization rules to prevent inadvertently applying additional rules and further changing the outcome. Instead, we create a new analyzer with a custom batch of optimization rules (line 4 - 8).

In these tests, we use a straightforward relation consisting of three attributes denoted by `'a`, `'b`, and `'c` respectively using the Spark SQL DSL syntax.

We will now look at the first test which checks whether a redundant skyline is really removed in more detail (line 13 - 25) First, we create a new skyline query (line 14 - 18) where dimension `'a` (minimized) is redundant since it appears twice (line 15 and 16). Subsequently, we call the optimizer and pass the (analyzed) query as a parameter (line 19). We also define the correct logical plan (where one instance of dimension `'a` is removed) which is also analyzed (line 20 - 23). Finally, we use the already provided function `comparePlans` to check whether the plans match (line 24).

There are three test cases for removing redundant skyline dimensions in total:

- Remove a single redundant skyline dimension (line 13 - 25; see above).
- Do not remove a skyline dimension if the type (`MIN/MAX/DIFF`) is different (line 27 - 40).
- Remove a skyline operator in its entirety if the list of skyline dimensions is empty (line 42 - 47).

Listing 6.2: Test cases for testing the removal of redundant skyline dimensions

```
1  class SkylineBasicSuite extends AnalysisTest{
2    val analyzer: Analyzer = getAnalyzer
3
4    object Optimize extends RuleExecutor[LogicalPlan] {
5      val batches = Batch("SkylineBasic", FixedPoint(1),
6        RemoveEmptySkylines,
7        RemoveRedundantSkylineDimensions) :: Nil
8    }
9
10   val testRelation: LocalRelation =
11     LocalRelation('a.int, 'b.int, 'c.int)
12
13   test("remove redundant skyline dimension") {
14     val query = testRelation.skyline(
15       SkylineDimension( 'a, SkylineMin ),
16       SkylineDimension( 'a, SkylineMin ),
17       SkylineDimension( 'b, SkylineMin )
18     )
19     val optimized = Optimize.execute(query.analyze)
20     val correctAnswer = testRelation.skyline(
21       SkylineDimension( 'a, SkylineMin ),
22       SkylineDimension( 'b, SkylineMin ),
23     ).analyze
24     comparePlans(optimized, correctAnswer)
25   }
26
27   test("do not remove pseudo-redundant dimension") {
28     val query = testRelation.skyline(
29       SkylineDimension( 'a, SkylineMin ),
30       SkylineDimension( 'a, SkylineMax ),
31       SkylineDimension( 'b, SkylineMin ),
32     )
33     val optimized = Optimize.execute(query.analyze)
34     val correctAnswer = testRelation.skyline(
35       SkylineDimension( 'a, SkylineMin ),
36       SkylineDimension( 'a, SkylineMax ),
37       SkylineDimension( 'b, SkylineMin ),
38     ).analyze
39     comparePlans(optimized, correctAnswer)
40   }
41
42   test("remove empty skyline") {
43     val query = testRelation.skyline().where('a > Literal(1))
44     val optimized = Optimize.execute(query.analyze)
45     val correctAnswer = testRelation.where('a > Literal(1)).analyze
46     comparePlans(optimized, correctAnswer)
47   }
48 }
```

6.1.3 Test Transformations of Single-Dimensional Skylines

Next, we test the transformations of single-dimensional skylines. The code for these test cases can be found in Listing 6.3. It utilizes a specialized helper function which can be found in Listing 6.4 and will be discussed after the general test cases.

We use an analyzer (line 2), a test relation (line 7 - 8), and an optimizer with only the single relevant optimization rule which transforms the skyline query to "pure" SQL queries (line 3 - 6).

In case of `MIN` and `MAX` skyline dimensions, we first apply the optimization via the custom optimizer (see, e.g., line 13) and then check the generated plan. This is done via a specialized function `checkSingleDimensionalMinMaxTransformation` (discussed shortly). This is necessary since the aggregate aliases are resolved to different reference numbers internally. The plans are then different according to the provided `comparePlans` function.

The test cases for `MIN` and `MAX` include:

- Transforming a single dimensional non-distinct skyline where the dimension is minimized (line 10 - 16)
- Transforming a single dimensional distinct skyline where the dimension is minimized (line 18 - 24)
- Transforming a single dimensional non-distinct skyline where the dimension is maximized (line 26; omitted for brevity)
- Transforming a single dimensional distinct skyline where the dimension is maximized (line 26; omitted for brevity)

The test cases also include two scenarios for single-dimensional skylines where the dimension is `DIFF`:

- Remove single-dimensional skylines containing only `DIFF` dimensions (line 28 - 34).
- Replace distinct skylines with a single `DIFF` dimension by `Deduplicate` with regards to the dimension to eliminate the skyline (line 36 - 43).

Listing 6.3: Test cases for testing the transformation of single-dimensional skyline queries

```
1  class SkylineSingleDimensionSuite extends AnalysisTest {
2    val analyzer: Analyzer = getAnalyzer
3    object Optimize extends RuleExecutor[LogicalPlan] {
4      val batches = Batch("SkylineBasic", FixedPoint(1),
5        RemoveSingleDimensionalSkylines) :: Nil
6    }
7    val testRelation: LocalRelation =
8      LocalRelation('a.int, 'b.int, 'c.int)
9
10   test("transform single dimensional non-distinct MIN skyline") {
11     val query = testRelation.skyline(
12       SkylineDimension('a.expr, SkylineMin) )
13     val optimized = Optimize.execute(query.analyze)
14     checkSingleDimensionalMinMaxTransformation(
15       optimized, min = true, distinct = false )
16   }
17
18   test("transform single dimensional distinct MIN skyline") {
19     val query = testRelation.skylineDistinct(
20       SkylineDimension('a.expr, SkylineMin) )
21     val optimized = Optimize.execute(query.analyze)
22     checkSingleDimensionalMinMaxTransformation(
23       optimized, min = true, distinct = true )
24   }
25
26   [...]
27
28   test("remove single dimensional non-distinct DIFF skyline") {
29     val query = testRelation.skyline(
30       SkylineDimension('a, SkylineDiff) )
31     val optimized = Optimize.execute(query.analyze)
32     val correctAnswer = testRelation.analyze
33     comparePlans(optimized, correctAnswer)
34   }
35
36   test("transform single dimensional distinct DIFF skyline") {
37     val query = testRelation.skylineDistinct(
38       SkylineDimension('a, SkylineDiff) )
39     val optimized = Optimize.execute(query.analyze)
40     val correctAnswer = Deduplicate(
41       'a :: Nil, testRelation ).analyze
42     comparePlans(optimized, correctAnswer)
43   }
44 }
```

Next, we will describe the `checkSingleDimensionalMinMaxTransformation` function which is responsible for checking whether the single dimensional transformations for `MIN` and `MAX` are correct. The code can be found in Listing 6.4. As already described above, this is necessary since the regular resolution of the analyzer will assign different reference ids to the aggregate aliases which produced wrong results for the predefined `comparePlans` function.

We use the Scala pattern matching to compare the plans. If no plan is matched, we set a flag (line 3) which is checked at the end of the function. The tests fail if and only if no correct plan was matched and the flag is consequently set (line 48). We distinguish between distinct and non-distinct skylines (line 4). The main differences are the local and global limits (line 6 - 7) which are only introduced in case of distinct skyline queries (see Section 5.8.3).

The rest of the pattern corresponds directly to the optimization described in Section 5.8.3. We match a filter (line 8 and 29) which extracts the result of a scala subquery (line 11 and 32). The aggregate in the scalar subquery depends on whether the dimension is minimized or maximized (line 14 and 35). This case is handled by a separate matching which depends on whether `Min` or `Max` appears in the query and whether the input parameter `min` was set (line 22 - 25 and line 42 - 45).

Listing 6.4: Helper function for testing the transformation of single-dimensional skylines

```
def checkSingleDimensionalMinMaxTransformation
( plan: LogicalPlan, min: Boolean, distinct: Boolean ): Unit = {
  var isFailedCheck = false
  if (distinct) {
    plan match {
      case GlobalLimit(Literal(1, IntegerType),
        LocalLimit(Literal(1, IntegerType),
          Filter( And(
              IsNotNull(_),
              EqualTo(_,
                ScalarSubquery( Aggregate(
                    _, Alias(
                      AggregateExpression(
                        minMax : AggregateFunction, _, _, _, _
                      ), _
                    ) :: Nil,
                    Project(_, _: LocalRelation)
                  ), _, _
            ) ) ),
            _: LocalRelation
        ) )
      ) => minMax match {
          case _: Min if min => isFailedCheck = false
          case _: Max if !min => isFailedCheck = false
          case _ => isFailedCheck = true }
      case _ => isFailedCheck = true }
  } else {
    plan match {
      case Filter( And(
          IsNotNull(_),
          EqualTo(_,
            ScalarSubquery( Aggregate(
                _, Alias(
                  AggregateExpression(
                    minMax : AggregateFunction, _, _, _, _
                  ), _
                ) :: Nil,
                Project(_, _: LocalRelation)
              ), _, _
        ) ) ),
        _: LocalRelation
      ) => minMax match {
          case _: Min if min => isFailedCheck = false
          case _: Max if !min => isFailedCheck = false
          case _ => isFailedCheck = true }
      case _ => isFailedCheck = true
  } }
  if (isFailedCheck) { fail([...]) }
}
```

6.1.4 Test Block-Nested-Loop Skyline Algorithm

Next, we test the implementation of the block-nested-loop skyline algorithm for complete skylines. The test cases belong to a single file but were split into two listings due to page size restrictions. The tests can be found in Listing 6.5 and Listing 6.6.

In these test cases, the main difference to the test cases already discussed above is that we do not check the parsed or transformed plan but rather whether the algorithm returns the correct result. For this purpose, we use the `checkAnswer` function which accepts a dataframe, a spark plan function, and an expected result as parameters.

To generate the input dataframes, we take a Scala sequence and then apply the `toDF` function. An example can be found in line 6 and line 8 of Listing 6.5. This is virtually the same for each test case with slightly altered data.

The test cases for the block-nested loop algorithm are split into two parts due to page size constraints and include:

- First part (Listing 6.5)
  - Test a skyline query with two minimized dimensions (line 5 - 17)
  - Test a skyline query with two maximized dimensions (line 19 - 31)
  - Test a skyline query with two maximized dimension and one `DIFF` dimension (line 33 - 47)
- Second part (Listing 6.6)
  - Test a non-distinct skyline query where two dimensions are minimized (line 1 - 14)
  - Test a distinct skyline query where two dimensions are minimized (line 16 - 43).

For the last test of the second part, additional care must be taken since there are two possibilities for the selection of correct tuples. We use a second test in a `catch` clause (line 30) that contains the second possible correct solution (line 32 - 41) since the result is deterministic regardless.

Listing 6.5: Test result of skyline queries using the block-nested-loop algorithm (part 1)

```
class SkylineBlockNestedLoopSuite extends SparkPlanTest
with SharedSparkSession {
  import testImplicits.{localSeqToDatasetHolder, newProductEncoder}

  test("basic min skyline using BlockNestedLoop") {
    val input = Seq( ("A", 1, 2.0), ("B", 3, 4.0), ("C", 4, 3.0) )
    checkAnswer(
      input.toDF("a", "b", "c"),
      (child: SparkPlan) => BlockNestedLoopSkylineExec(
        SkylineIsNotDistinct,
        'b.smin :: 'c.smin :: Nil,
        requiredChildDistributionExpressions = Some(Nil),
        isIncompleteSkyline = false,
        child
      ),
      Seq(("A", 1, 2.0)).map(Row.fromTuple))
  }

  test("basic max skyline using BlockNestedLoop") {
    val input = Seq( ("A", 1, 2.0), ("B", 3, 4.0), ("C", 4, 3.0) )
    checkAnswer(
      input.toDF("a", "b", "c"),
      (child: SparkPlan) => BlockNestedLoopSkylineExec(
        SkylineIsNotDistinct,
        'b.smax :: 'c.smax :: Nil,
        requiredChildDistributionExpressions = Some(Nil),
        isIncompleteSkyline = false,
        child
      ),
      Seq(("B", 3, 4.0), ("C", 4, 3.0)).map(Row.fromTuple))
  }

  test("basic diff skyline using BlockNestedLoop") {
    val input = Seq( ("A", 1, 2.0, 1), ("B", 3, 4.0, 2),
                     ("C", 4, 3.0, 3) )
    checkAnswer(
      input.toDF("a", "b", "c", "d"),
      (child: SparkPlan) => BlockNestedLoopSkylineExec(
        SkylineIsNotDistinct,
        'b.smax :: 'c.smax :: 'd.sdiff :: Nil,
        requiredChildDistributionExpressions = Some(Nil),
        isIncompleteSkyline = false,
        child
      ),
      Seq(("A", 1, 2.0, 1), ("B", 3, 4.0, 2), ("C", 4, 3.0, 3))
        .map(Row.fromTuple))
  }
```

Listing 6.6: Test result of skyline queries using the block-nested-loop algorithm (part 2)

```
1    test("non-distinct skyline using BlockNestedLoop") {
2      val input = Seq( ("A", 1, 2.0), ("B", 1, 2.0), ("C", 4, 3.0) )
3
4      checkAnswer(
5        input.toDF("a", "b", "c"),
6        (child: SparkPlan) => BlockNestedLoopSkylineExec(
7          SkylineIsNotDistinct,
8          'b.smin :: 'c.smin :: Nil,
9          requiredChildDistributionExpressions = Some(Nil),
10         isIncompleteSkyline = false,
11         child
12       ),
13       Seq(("A", 1, 2.0), ("B", 1, 2.0)).map(Row.fromTuple))
14   }
15
16   test("distinct skyline using BlockNestedLoop") {
17     val input = Seq( ("A", 1, 2.0), ("B", 1, 2.0), ("C", 4, 3.0) )
18
19     try {
20       checkAnswer(
21         input.toDF("a", "b", "c"),
22         (child: SparkPlan) => BlockNestedLoopSkylineExec(
23           SkylineIsDistinct,
24           'b.smin :: 'c.smin :: Nil,
25           requiredChildDistributionExpressions = Some(Nil),
26           isIncompleteSkyline = false,
27           child
28         ),
29         Seq(("B", 1, 2.0)).map(Row.fromTuple))
30     } catch {
31       case _: TestFailedException =>
32         checkAnswer(
33           input.toDF("a", "b", "c"),
34           (child: SparkPlan) => BlockNestedLoopSkylineExec(
35             SkylineIsDistinct,
36             'b.smin :: 'c.smin :: Nil,
37             requiredChildDistributionExpressions = Some(Nil),
38             isIncompleteSkyline = false,
39             child
40           ),
41           Seq(("A", 1, 2.0)).map(Row.fromTuple))
42     }
43   }
44 }
```

6.1.5 Test Distributed Complete Skyline Algorithm

The test cases for the distributed algorithm are based on the ones for block-nested-loop. We only need to change the logical plan which needs to accommodate the more complex distributed processing.

The test cases can be found in Listing 6.7 and Listing 6.8. The listings contain two parts of the same code fragment which was split due to page size constraints.

For a description of the test cases, we refer to Section 6.1.4 since they are the same except for the changed and more complex query plan.

Listing 6.7: Test result of skyline queries using the distributed complete algorithm (part 1)

```
class SkylineDistributedSuite extends SparkPlanTest
with SharedSparkSession {
  test("basic min skyline using BlockNestedLoop") {
    val input = Seq( ("A", 1, 2.0), ("B", 3, 4.0), ("C", 4, 3.0) )
    checkAnswer(
      input.toDF("a", "b", "c"),
      (child: SparkPlan) => BlockNestedLoopSkylineExec(
        SkylineIsNotDistinct, 'b.smin :: 'c.smin :: Nil,
        requiredChildDistributionExpressions = Some(Nil),
        isIncompleteSkyline = false,
        BlockNestedLoopSkylineExec(
          SkylineIsNotDistinct, 'b.smin :: 'c.smin :: Nil,
          None, false, child
      ) ),
      Seq(("A", 1, 2.0)).map(Row.fromTuple))
  }
  test("basic max skyline using BlockNestedLoop") {
    val input = Seq( ("A", 1, 2.0), ("B", 3, 4.0), ("C", 4, 3.0) )
    checkAnswer(
      input.toDF("a", "b", "c"),
      (child: SparkPlan) => BlockNestedLoopSkylineExec(
        SkylineIsNotDistinct, 'b.smax :: 'c.smax :: Nil,
        requiredChildDistributionExpressions = Some(Nil),
        isIncompleteSkyline = false,
        BlockNestedLoopSkylineExec(
          SkylineIsNotDistinct, 'b.smax :: 'c.smax :: Nil,
          None, false, child
      ) ),
      Seq(("B", 3, 4.0), ("C", 4, 3.0)).map(Row.fromTuple))
  }
  test("basic diff skyline using BlockNestedLoop") {
    val input = Seq( ("A", 1, 2.0, 1), ("B", 3, 4.0, 2),
                     ("C", 4, 3.0, 3) )
    checkAnswer(
      input.toDF("a", "b", "c", "d"),
      (child: SparkPlan) => BlockNestedLoopSkylineExec(
        SkylineIsNotDistinct,
        'b.smax :: 'c.smax :: 'd.sdiff :: Nil,
        requiredChildDistributionExpressions = Some(Nil),
        isIncompleteSkyline = false,
        BlockNestedLoopSkylineExec(
          SkylineIsNotDistinct,
          'b.smax :: 'c.smax :: 'd.sdiff :: Nil,
          None, false, child
      ) ),
      Seq(("A", 1, 2.0, 1), ("B", 3, 4.0, 2), ("C", 4, 3.0, 3))
        .map(Row.fromTuple))
  }
```

Listing 6.8: Test result of skyline queries using the distributed complete algorithm (part 2)

```
1    test("non-distinct skyline using BlockNestedLoop") {
2      val input = Seq( ("A", 1, 2.0), ("B", 1, 2.0), ("C", 4, 3.0) )
3      checkAnswer(
4        input.toDF("a", "b", "c"),
5        (child: SparkPlan) => BlockNestedLoopSkylineExec(
6          SkylineIsNotDistinct, 'b.smin :: 'c.smin :: Nil,
7          requiredChildDistributionExpressions = Some(Nil),
8          isIncompleteSkyline = false,
9          BlockNestedLoopSkylineExec(
10           SkylineIsNotDistinct,
11           'b.smin :: 'c.smin :: Nil,
12           requiredChildDistributionExpressions = None,
13           isIncompleteSkyline = false, child
14       ) ),
15       Seq(("A", 1, 2.0), ("B", 1, 2.0)).map(Row.fromTuple))
16   }
17
18   test("distinct skyline using BlockNestedLoop") {
19     val input = Seq( ("A", 1, 2.0), ("B", 1, 2.0), ("C", 4, 3.0) )
20     try {
21       checkAnswer(
22         input.toDF("a", "b", "c"),
23         (child: SparkPlan) => BlockNestedLoopSkylineExec(
24           SkylineIsDistinct, 'b.smin :: 'c.smin :: Nil,
25           requiredChildDistributionExpressions = Some(Nil),
26           isIncompleteSkyline = false,
27           BlockNestedLoopSkylineExec(
28             SkylineIsDistinct, 'b.smin :: 'c.smin :: Nil,
29             requiredChildDistributionExpressions = None,
30             isIncompleteSkyline = false, child
31         ) ),
32         Seq(("B", 1, 2.0)).map(Row.fromTuple))
33     } catch {
34       case _: TestFailedException =>
35         checkAnswer(
36           input.toDF("a", "b", "c"),
37           (child: SparkPlan) => BlockNestedLoopSkylineExec(
38             SkylineIsDistinct, 'b.smin :: 'c.smin :: Nil,
39             requiredChildDistributionExpressions = Some(Nil),
40             isIncompleteSkyline = false,
41             BlockNestedLoopSkylineExec(
42               SkylineIsDistinct, 'b.smin :: 'c.smin :: Nil,
43               requiredChildDistributionExpressions = None,
44               isIncompleteSkyline = false, child
45           ) ),
46           Seq(("A", 1, 2.0)).map(Row.fromTuple))
47 } } }
```

6.1.6 Test Distributed Incomplete Skyline Algorithm

For the last tests, we give the test cases for the distributed skyline in Listing 6.9. The tests are similar to ones for the distributed processing but the test data needs to be modified to allow for missing values.

Since this algorithm is very similar to the "regular" distributed algorithm, we limit ourselves to cases where dimensions are missing. To create a dataframe with missing values from a Scala sequence, we use `Some(...)` and `None`. Every `None` corresponds with a missing value while every non-missing value must be wrapped in `Some(...)` to ensure type compatibility with the `None` values.

We include three test cases in our implementation:

- Test a basic skyline query for an incomplete distributed algorithm where two dimensions are minimized (line 3 - 15).
- Test an extended incomplete distributed skyline query by specifying three dimensions (line 16 - 32).
- Test the detection of cyclic dominance relationships in the dataset (line 33 - 45). For this, we use the example of cyclic dependencies/loss of transitivity from Section 3.2. The expected result of this query contains no tuples (line 45).

Listing 6.9: Test result of skyline queries using the distributed incomplete algorithm

```
class SkylineIncompleteSuite extends SparkPlanTest
with SharedSparkSession {
  test("basic min skyline using incomplete distributed skylines") {
    val input = Seq( ("A",Some(1),Some(2.0)),("B",None,Some(4.0)),
                     ("C",Some(4),None) )
    checkAnswer(
      input.toDF("a", "b", "c"),
      (child: SparkPlan) => IncompleteSkylineExec(
        SkylineIsNotDistinct, 'b.smin :: 'c.smin :: Nil,
        BlockNestedLoopSkylineExec(
          SkylineIsNotDistinct, 'b.smin :: 'c.smin :: Nil,
          Some(IsNull('b).expr :: IsNull('c).expr :: Nil),
          true, child
      ) ), Seq(("A", 1, 2.0)).map(Row.fromTuple))
  }
  test("extended min skyline using incomplete distributed skylines"){
    val input = Seq(
      ("A", None, Some(4.0), None), ("B", None, Some(5.0), None),
      ("C", Some(4), None, None), ("D", Some(5), None, None),
      ("E", None, None, Some(4.0)), ("F", None, None, Some(3.0)) )
    checkAnswer(
      input.toDF("a", "b", "c", "d"),
      (child: SparkPlan) => IncompleteSkylineExec(
        SkylineIsNotDistinct, 'b.smin :: 'c.smin :: 'd.smin :: Nil,
        BlockNestedLoopSkylineExec(
          SkylineIsNotDistinct,
          'b.smin :: 'c.smin :: 'd.smin :: Nil,
          Some(IsNull('b).expr :: IsNull('c).expr ::
               IsNull('d).expr :: Nil), true, child
      ) ), Seq( ("A", null, 4.0, null), ("C", 4, null, null),
                ("F", null, null, 3.0)).map(Row.fromTuple) )
  }
  test("eliminate cyclic dominance in incomplete skylines") {
    val input = Seq(
      ("A", Some(1),None,Some(10)), ("B",Some(3),Some(2),None),
      ("C", None,Some(5),Some(3)) )
    checkAnswer(
      input.toDF("a", "b", "c", "d"),
      (child: SparkPlan) => IncompleteSkylineExec(
        SkylineIsNotDistinct, 'b.smin::'c.smin::'d.smin :: Nil,
        BlockNestedLoopSkylineExec(
          SkylineIsNotDistinct, 'b.smin::'c.smin::'d.smin :: Nil,
          Some(IsNull('b).expr :: IsNull('c).expr ::
               IsNull('d).expr :: Nil), true, child
      ) ), Seq().map(Row.fromTuple))
} }
```

6.2 System Testing

By system testing we want to check whether the implementation is able to load data from a data source, process it, and return the correct skyline. This is a full test which checks all parts of the implementation from the parser to the physical plan execution node.

For these tests, we need to find the "correct" results for each dataset first. We utilize the translations to "plain" SQL which are described in Section 2.4.1 and Section 3.2.2 for this. We simply execute both the skyline query and the "plain" SQL query and then check whether the tuples in both results match.

We assume for this method that the implementation of "plain" SQL queries in Spark is correct. Otherwise, there may be false positives or false negatives to these tests. This assumption is reasonable since our implementation also builds on the existing parts of Spark SQL. If there is an error unrelated to the actual skyline query implementation, there is still a high chance that many skyline queries will also not compute the correct results. It follows that testing the skyline queries is only ever sensible under the assumption that the underlying Spark SQL system is working correctly.

To simplify looking at the test results, we use a Jupyter notebook to execute the tests and display their results. We demonstrate the tests based on a single test case as an example. All other test cases work analogously by changing the datasets and dimensions. The code for the test case which we will discuss here can be found in Listing 6.10.

We use the table `airbnb_test` from the `tests` database, the minimized dimension `price`, and `accommodates` which is maximized.

The reference query which yields the "correct" result to compare against can be found in line 1 - 11. The DataFrame `df_airbnb_reference_2d` holds the result of the reference query. In this test case, we use the query found in line 13 - 18 and store the result in the dataframe `df_airbnb_bnl_2d`. By using the `BNL` keyword we ensure that we are testing the block-nested-loop algorithm.

Subsequently, we check whether there is a tuple which exists in only one of the result dataframes. The function `exceptAll` returns all tuples that are in the first DataFrame but not in the second. Using this in both directions, we can ensure that both results contain the same tuples if both `exceptAll` comparisons are empty (line 20 - 24 and line 25 - 29). We can determine whether the result of the `exceptAll` comparisons is empty by taking the first element as a list using `head(1)` (line 22 and 27) and then checking whether the length of the retrieved list is 0 (line 20 and 23 as well as line 25 and 28).

Listing 6.10: System test of a 2-dimensional skyline query

```
df_airbnb_reference_2d = sqlContext.sql("""
SELECT * FROM tests.airbnb_test AS o WHERE NOT EXISTS(
  SELECT * FROM tests.airbnb_test AS i WHERE
    i.price <= o.price AND
    i.accommodates >= o.accommodates
    AND (
      i.price < o.price OR
      i.accommodates > o.accommodates
    )
) ORDER BY id
""")

df_airbnb_bnl_2d = sqlContext.sql("""
SELECT * FROM tests.airbnb_test SKYLINE OF BNL
  price MIN,
  accommodates MAX
ORDER BY id
""")

display(len(
  df_airbnb_bnl_2d.exceptAll(
    df_airbnb_reference_2d).head(1)
  ) == 0
)
display(len(
  df_airbnb_reference_2d.exceptAll(
    df_airbnb_bnl_2d).head(1)
  ) == 0
)
```

6.3 Benchmarks

In this section, we analyze how our implementation performs on test data to gauge its usefulness in the real world. To achieve this, we execute various skyline queries on synthetic and real-world datasets.

Our main metrics are the time needed to retrieve the result of the query and the total memory consumption across the entire benchmarking system. We will use a standalone and a clustered environment to execute the queries and emulate two different use cases. Both environments will be discussed in more detail in Section 6.3.1.

The input used for benchmarking consists of five different input datasets. Four datasets are taken from the real world and have, in part, also been used to benchmark other skyline processing systems. The fifth dataset is synthetically generated which allows us

to easily sale the size of the dataset and create different scenarios for benchmarking. The datasets will be discussed in detail in Section 6.3.2.

For the real-world datasets, we will tweak the following parameters to measure their impact on the performance:

- Number of skyline dimensions
- Number of executor nodes (clustered environment only)
- Total number of executor cores (standalone environment only)

The synthetic dataset can be scaled indefinitely and can be used to easily vary the size of the dataset. Due to the size of the synthetic dataset, we will only the corresponding skyline queries in clustered environments. It follows that we will tweak the following parameters for the synthetic dataset:

- Number of skyline dimensions
- Number of executor nodes
- Size of the dataset

6.3.1 Benchmarking Environments

For benchmarking purposes, we use two different environments which represent two potential uses of skyline queries. The standalone environment emulates running skyline queries on a local machine while the clustered environment mimics the behavior of industrial applications such as recommendation engines running on distributed systems.

6.3.1.1 Standalone Environment

The standalone environment serves as a test platform for executing the skyline queries on a relatively recent personal computer. Its specifications can be found in Table 6.1.

Component	Specification	Info
CPU	AMD Ryzen 5 3600	6 (12[1]) cores
RAM	32GB DDR4	
Disk	Samsung 970 EVO NVMe M.2 SSD	1 TB
OS	Ubuntu 20.04	

[1] number of virtual cores achieved using hyperthreading

Table 6.1: Standalone Environment Specification

6.3.1.2 Clustered Environment

Due to the distributed nature of Spark, we also test the implementation on a distributed system. This allows the Spark skyline queries to make use of the distributed nature of the cluster and compute parts of the results (local skylines) in parallel.

We use the *little big data* (lbd) cluster provided by the High Performance Computing team of the TU Wien. It provides an environment which hosts a multitude of distributed processing applications such as Hive, Cassandra, MongoDB, Kafka, and Spark 2 (different instance than the Spark 3 version used for this thesis). It uses a Cloudera platform based on Hadoop and consists of 2 name nodes and 18 datanodes which are governed by a single (additional) admin node.

The specification for each of the 20 non-admin nodes can be found in Table 6.2.

Component	**Specification**	**Info**
CPU	2x Xeon E5-2650v4	24 (48[1]) cores **each**
RAM	256 GB	
Disk	4 disks with 4 TB each	16 TB
Connections	10 Gb/s Ethernet connections	

[1] number of virtual cores achieved using hyperthreading

Table 6.2: Standalone Environment Specification

Across the system, this adds up to the following total available resources:

- 864 worker cores (excluding hyperthreading)
- 4.5 TB of RAM
- 288 TB disk capacity

The cluster also has a dedicated ZFS file storage with a capacity of 300 TB which is separate from the rest of the available resources.

The Hadoop File system (HDFS) running on the cluster is configured as follows:

- Hadoop 3
- 128 MiB per block
- replication factor of 3

For storing the data of these benchmarks we use Hive which loads the data from .csv files. The following specifications apply to the instance of Hive on the lbd cluster:

- Hive 2.1.1
- No headers in source files
- Manual schema creation

We omit the header from our input (.csv) files since Spark may not interpret the header correctly when reading from a Hive source. This makes it necessary to create the schema manually which is done before the data is imported from the .csv file.

6.3.2 Queries and Datasets

We use query strings for benchmarking the skyline queries. This allows us to formulate the skyline queries in a straightforward way and to write the corresponding reference queries in plain SQL. As a consequence, it is also possible to generate the queries automatically.

The "regular" skyline queries can be found in Listing 6.11. It matches the standard formulation of skyline queries (Listing 2.1) with the additional keywords `COMPLETE` (for the complete distributed algorithm) and `BNL` (for the block-nested-loop algorithm). Since we do not use `DISTINCT` skylines in the benchmarks, the keyword was omitted here.

Listing 6.11: Original skyline query [BKS01, Ede09]

```
1 SELECT column_list FROM relation AS r
2 SKYLINE OF [ COMPLETE | BNL ] a_1 MIN, ..., a_j MIN, a_{j+1} MAX, ..., a_k
    MAX, a_{k+1} DIFF, ..., a_m DIFF;
```

For the reference queries, we use the standard translation methods as discussed in Section 2.4.1 and Section 3.2.2 to translate the skyline queries to "plain" SQL.

Next, we look at the datasets and skyline dimensions which we utilize in the benchmarks. For each table, we list all skyline dimensions, whether they are minimized or maximized, and a short description which offers some context on how to interpret the skyline query where applicable. Additionally, we also list the "primary key" attribute(s) for each table. These attributes are unique for each tuple in the table and can therefore be used to identify the tuple. Spark, however, cannot distinguish between "regular" attributes and primary keys.

The first dataset contains accommodations offered via the Airbnb platform obtained via the Inside Airbnb platform [Ins]. Datasets can be downloaded for each city or region and always contain data for a single month. This dataset is a realistic application of skyline queries since it can be used to find the "best" accommodations. An overview over the primary keys and dimensions can be found in Table 6.3.

Since the original dataset is not complete (with regards to the primary keys and skyline dimensions), we create two different datasets where the first one contains the original (incomplete) data while all incomplete tuples were removed in the second one which is therefore complete with regards to the skyline dimensions. The datasets have 1193465 and 820698 tuples respectively.

Dimension	Type	Description
id	KEY	identification number
price	MIN	price for renting
accommodates	MAX	(max) number of accommodated people
bedrooms	MAX	number of bedrooms
beds	MAX	number of beds
number_of_reviews	MAX	number of reviews
review_scores_rating	MAX	total review score ratings (all categories)

Table 6.3: Skyline Dimensions Airbnb dataset

The Fuel Economy dataset [Fue] contains estimated fuel consumption statistics for cars. This is also a good example for a real-world application of skyline queries as it emulates the use case of finding the "best" cars by their optimal fuel consumption. Each car is identified by its make and model. An overview over the primary keys and the skyline dimensions can be found in Table 6.4.

Since this is an incomplete dataset with regards to the skyline dimensions, we also generate a second, complete dataset from it. These datasets have 22276 and 22490 tuples respectively.

Dimension	Type	Description
make	KEY	make of the car
model	KEY	model of the car
fuelCost08	MIN	annual fuel costs
barrels08	MIN	annual fuel consumption (barrels)
city08	MAX	city miles-per-gallon
highway08	MAX	highway miles-per-gallon
comb08	MAX	combined miles-per-gallon
combinedCD	MIN	combined gasoline consumption in charge-depleting mode

Table 6.4: Skyline Dimensions Fueleconomy dataset

Next, we look at the COIL 2000 dataset [COI] which comes from the census of the united states. It contains data about various households which are identified by their IDs. An overview over the primary key and skyline dimensions can be found in Table 6.5. This is a complete dataset by default. It follows that the variant obtained by removing incomplete tuples is identical. Both variants contain 5822 tuples.

Dimension	Type	Description
ID	KEY	identification number
MOSHOOFD	MAX	Customer Main Type
MGODRK	MIN	Roman Catholic
MGODPR	MIN	Protestant
MGODGE	MAX	No religion
MRELGE	MAX	Married
MRELOV	MAX	Other Relation

All decision about minimization and maximization were done arbitrarily for this table. They are not in any way indicators whether minimization or maximization is desirable in the real world.

Table 6.5: Skyline Dimensions COIL 2000 dataset

The last and smallest of the real-world datasets is the dataset about current NBA players [NBA] which contains statistics about them. It was obtained from the official NBA webpage [NBA] by converting the data to a .csv file. Every player has a unique PLAYER_ID which is used as the primary key. Here, the skyline queries serve to find the "best" players with regards to specific performance data. An overview over the primary keys and skyline dimensions can be found in Table 6.6. This is, again, a complete dataset by default which means that the complete and incomplete variant of the data are identical. Both datasets contain 446 tuples.

Dimension	Type	Description
PLAYER_ID	KEY	identification number
W	MAX	number of Wins
L	MIN	number of Losses
W_PCT	MAX	win percentage
FGM	MAX	Field Goals Made
FGA	MAX	Field Goals Attempted
FG_PCT	MAX	Field Goals Percentage

Table 6.6: Skyline Dimensions NBA dataset

The synthetic dataset is based on the DSB benchmark which is commonly used to generate both data and queries for benchmarking [DCGN21]. For this thesis, we only make use of the data generating part of the benchmarking suite and devise the queries manually. For the benchmarks, the `store_sales` table of the generated data was selected. Each unique sale can be identified by its `item_sk` and `ticket_number`. An overview over the primary keys and skyline dimensions can be found in Table 6.7.

For generating the table, the source code can be obtained from GitHub [DSB]. For this thesis we use only parts of the generated data such that the table contains exactly 1 million, 2 million, 5 million and 10 million tuples respectively. The same goes for the complete variant of the dataset where we select only complete tuples. It follows that the complete and incomplete datasets contain the same amount of tuples but not necessarily the same tuples.

Dimension	Type	Description
ss_item_sk	KEY	stock item identifier
ss_ticket_number	KEY	ticket number identifier
ss_quantity	MAX	quantity purchased in sale
ss_wholesale_cost	MIN	wholesale cost
ss_list_price	MIN	list price
ss_sales_price	MIN	sales price
ss_ext_discount_amt	MAX	(total) amount of discount given
ss_ext_sales_price	MIN	sum of sales price

Table 6.7: Skyline Dimensions DST (store sales) dataset

6.3.3 Test Setup

In this section, we discuss the setup which we will use to run the benchmarks. This includes both starting the actual query and getting the relevant results for further processing.

We use `./spark-sql` to load the query strings from .sql files using the command line option `-f`. This allows us to directly work with the queries without having to embed them into a e.g. a python script.

6.3.3.1 Standalone Environment

For the standalone cluster we use a local instance of a hive database to directly store the data on disk. The following parameters apply to all queries issued in standalone mode:

- `--conf spark.sql.catalogImplementation=hive`
  - Use Hive as the catalog implementation to properly load the benchmarking dataset from the disk.
- `--conf spark.executor.processTreeMetrics.enabled=true`
  - Enable metrics which allow additional data like peak memory consumption to be extracted from the logs via the history server. This is used to obtain additional information while benchmarking.

- `--conf spark.executor.metrics.pollingInterval=10`
  - Shorten the interval in which the metrics are pulled by Spark. This has been observed to vastly increase the accuracy of the memory benchmarks. It also allows the memory consumption to be recorded even for queries on small datasets.

Since we always have one node, we vary the number of executor cores instead. Here, we note that one node must always be used for the driver. With our limit of 6 available cores, we use a maximum of 5 cores as executor cores. The number of cores is governed by:

- `--total-executor-cores ${nodes}`

6.3.3.2 Clustered Environment

In the clustered environment we use YARN to govern the execution of the benchmarks. We use up to 20 cores as executor cores per node and use up to 10 executors. We use the following settings on the server:

- `--deploy-mode client`
  - Use the current client process as driver. This increases the load on the name node but allows verbose outputs and easier termination of queries.
- `--conf spark.sql.catalogImplementation=hive`
  - Use Hive as the catalog implementation to properly load the benchmarking dataset from the Hive server.
- `--executor-cores 20`
  - Set the number of cores per node to 20.
- `--conf spark.executor.processTreeMetrics.enabled=true`
  - Enable metrics which allow additional data like peak memory consumption. This is used to obtain additional information while benchmarking. Data extraction, in this case, is done by copying the logs to a compatible Spark 3 instance and starting a history server.
- `--conf spark.executor.metrics.pollingInterval=10`
  - Shorten interval in which the metrics are pulled by Spark. This has been observed to vastly increase the accuracy of the memory benchmarks. It also allows the memory consumption to be recorded even for queries on small datasets. For some very long reference queries, this value was increased.
- `--conf spark.executor.heartbeatInterval=1000`
  - Set the interval of heartbeats to ensure that all queries are logged. For some very long reference queries, this value was increased.

In this environment, we use YARN to specify the number of executors to be used in each benchmark. The total number of executors is governed by:

- `--num-executors ${executors}`

6.3.3.3 Result Extraction

The results are extracted using a Python script which reads the application id from each job and then obtains the data from the history server via its REST API. Application IDs can be found in the output of each submitted Spark query and extracted using the following code that uses a regular expression to find the application id in the output file:

```
app_id = findall(r"Application Id: (.*)", data)
```

An excerpt from the script can be found in Listing 6.12. Assuming we have already extracted the `app_id` from the output, we first get the application data from the history server (line 1 - 2). Then we parse the JSON (line 4) and extract the duration (in milliseconds) which is converted to seconds (line 10). After that, we use the rest API to get the data from all executors for the given application id (line 14 - 16). We again parse the JSON response (line 19) and extract the on-heap and off-heap memory consumption (line 23 - 26). The total memory consumption is the sum of all on-heap and off-heap memory consumptions. Finally, we append the data as a tuple to a list of tuples which serves as the result of collecting the benchmarking data.

Listing 6.12: Algorithm for extracting the benchmark data from a history server

```
response =
  requests.get(f"{history_server}/api/v1/applications/{app_id[0]}")
if response.status_code == 200:
  response_json = response.json()
  if 'attempts' in response_json:
    for attempt in response_json['attempts']:
      if 'duration' in attempt and
                'completed' in attempt and
                attempt['completed'] == True:
          times = [float(attempt['duration']) / 1000]
      else:
          times = [-1]

response = requests.get(
  f"{history_server}/api/v1/applications/{app_id[0]}/executors"
)

if response.status_code == 200:
  response_json = response.json()
  for executor in response_json:
    if 'peakMemoryMetrics' in executor:
      peak_memory_metrics = executor['peakMemoryMetrics']
      if 'JVMHeapMemory' in peak_memory_metrics:
        total_memory += peak_memory_metrics['JVMHeapMemory']
      if 'JVMOffHeapMemory' in peak_memory_metrics:
        total_memory += peak_memory_metrics['JVMOffHeapMemory']

result_list.append(( filename_split[0], filename_split[1],
  "skyline", filename_split[4], filename_split[3],
  filename_split[2], times[0], total_memory) )
```

6.3.4 Benchmarking Results

In this section, we look at the results of the benchmarks in both environments introduced in Section 6.3.1.1 (standalone environment) and Section 6.3.1.2 (clustered environment).

Due to the vast amount of data involved in these benchmarks, we will look use plots to visualize the results.

For each environment, we will look at how the execution time changes depending on the number of skyline dimensions as well as the available processing resources such as total number of cores or number of executors. Additionally, we will look at how the size of the dataset influences the execution time for each environment. On the clustered environment, we will also discuss how the memory consumption scales with the number of dimensions and the number of nodes.

The memory consumption we will measure as part of these benchmarks is equal to the total on-heap and off-heap memory across all executors. It is therefore to be expected that a higher number of executors will lead to a higher total memory consumption. We also note that memory consumption is equivalent to **peak** memory consumption of the node, i.e., the highest memory consumption that was measured across the entire lifetime of the executor.

6.3.4.1 Standalone Environment

We will first take a look at the results from the standalone environment. Here, we have only applied the real-world dataset since the synthetic dataset is too big to be reasonably executed locally for its biggest instances.

For all results note that there is a delay of about 4 to 6 seconds for each and every query executed on the system. The execution time can therefore not fall below that threshold.

6.3.4.1.1 Real-Wold Dataset

First, we discuss how the number of dimensions influences the time needed to execute a skyline query. For this purpose, we look at the biggest real-world dataset (Inside Airbnb). The results for the complete and incomplete algorithms can be found in Figure 6.1 and Figure 6.2 respectively.

For both datasets and all algorithms, the number of dimensions does correlate directly to the execution time needed. The more dimensions the more comparisons are needed and the longer the execution time becomes. Furthermore, it is notable that in Figure 6.1 the time penalty introduced by additional dimensions is lower for the complete distributed algorithm than it is for the block-nested-loop algorithm and the distributed incomplete algorithm. The custom skyline algorithms are faster than the reference queries using “plain” SQL.

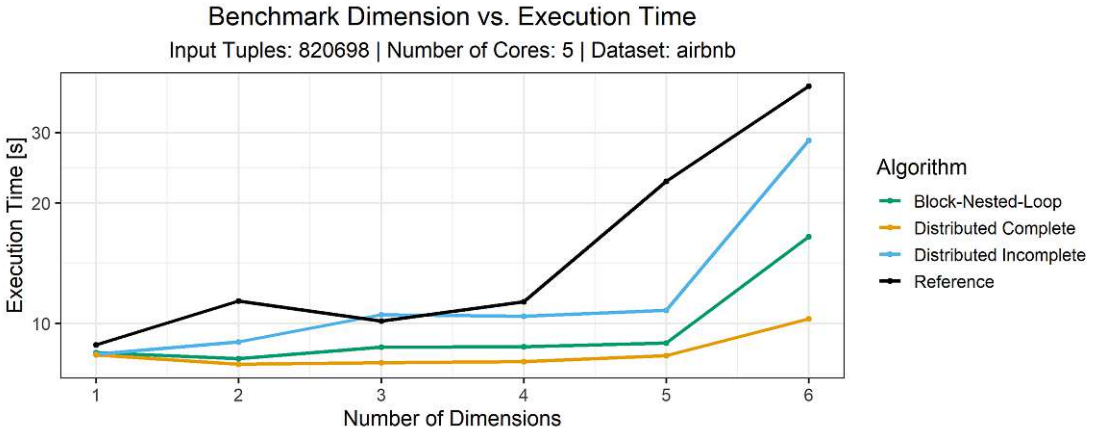

Figure 6.1: Number of dimensions vs. execution time on the complete Inside Airbnb dataset

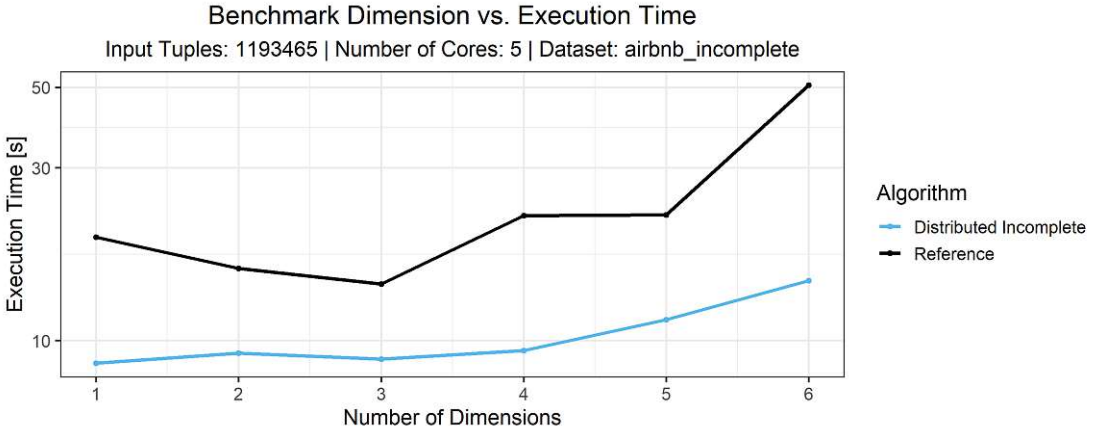

Figure 6.2: Number of dimensions vs. execution time on the incomplete Inside Airbnb dataset

Next, we want to investigate how the number of executor cores influences the time needed to execute the queries. The test results can be found in Figure 6.3 and Figure 6.4 for complete and incomplete datasets respectively.

It is most notable that the execution time goes down for the distributed complete algorithm and the reference queries. This is due to the fact that both variants can make good use of more cores to calculate partial data in parallel. In case of the reference solution, this is done by the default optimizations of Spark SQL. The distributed complete algorithm uses the cores to compute the local skylines more effectively in parallel.

For the block-nested-loop algorithm the times stay roughly the same. This is due to the fact that this algorithm is not distributed and can therefore only compute the global skyline in one step. All deviations can be attributed to jitter.

The results for the incomplete distributed algorithm are more peculiar since the execution time increases with the number of nodes. Similar to the block-nested-loop algorithm, it can only compute the entire result at once since distribution cannot be achieved through grouping by `null` values. The increase in execution time may therefore be at least partially be attributed to synchronization penalties or fluctuations in node assignment.

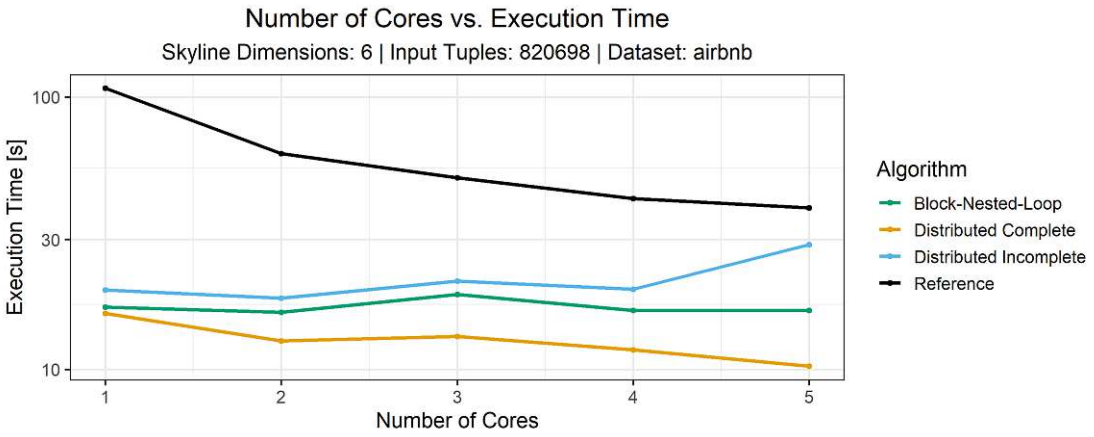

Figure 6.3: Number of nodes vs. execution time on the complete Airbnb dataset

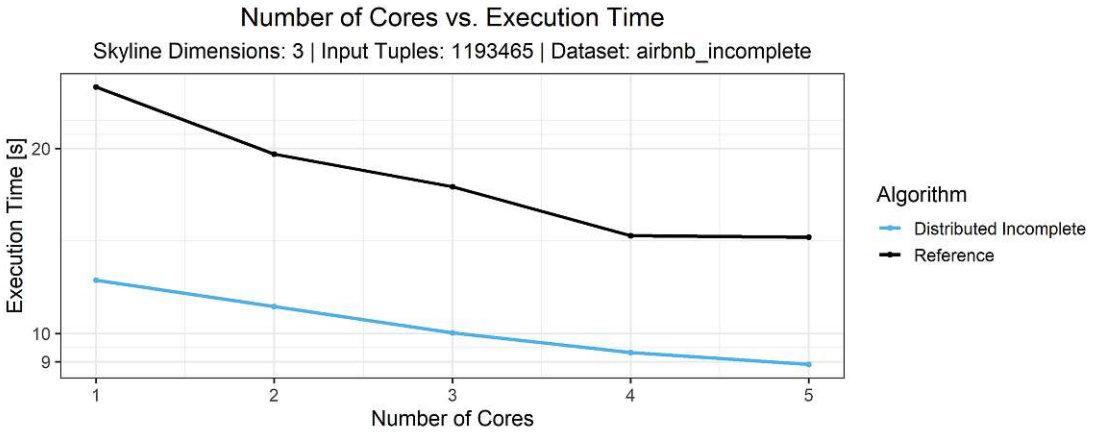

Figure 6.4: Number of nodes vs. execution time on the incomplete Airbnb dataset

We also look at how the execution time scales with the size for the dataset for a constant number of dimensions and nodes. The results can be found in Figure 6.5. Note that the samples for this plot come from different datasets (NBA, Fuel Economy, COIL 2000, and Inside Airbnb) which may introduce some inaccuracies.

It is noticeable that the advantage of the distributed algorithms becomes more apparent the bigger the dataset is. For smaller datasets, non-distributed algorithms or even the reference algorithms may be faster than their distributed variants. In this figure, it is most noticeable that the block-nested-loop algorithm is as good as the distributed algorithm for the smaller datasets while it is greatly outperformed for the bigger dataset.

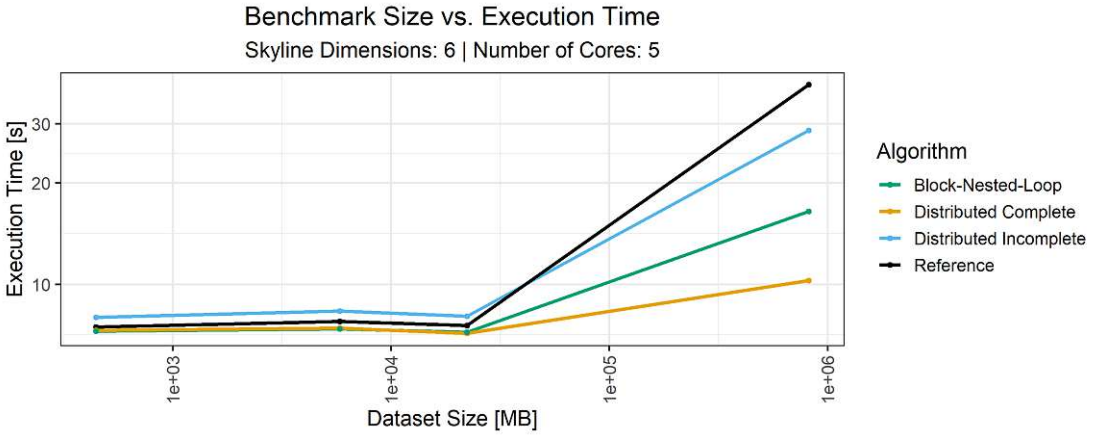

Figure 6.5: Dataset size vs. execution time on the complete real-world datasets

6.3.4.2 Cluster Environment

In this section, we will look at the benchmarking results from the clustered environment for both the real-world and the synthetic dataset. Note that the cluster also imposes a minimum delay of about 6 seconds for every query executed by the skyline.

6.3.4.2.1 Real-Wold Dataset

We again investigate the influence that the number of skyline dimensions has on the execution time of the queries. The test results can be found in Figure 6.6 and Figure 6.7 for the complete and incomplete variant of the Inside Airbnb dataset respectively.

Here, we can see that the number of dimensions still directly correlates with the execution time. we see that the number of dimensions directly correlates with the execution time needed with the exception of the 2-dimensional and 3-dimensional skyline query of the incomplete reference solution which have lower execution times than their predecessor points in the graph.

We observe that the custom algorithms are faster than the reference solution in all cases. For the complete algorithms, the distributed complete approach is faster than the block-nested-loop approach which, in turn, is faster than the distributed incomplete algorithm. This can be attributed to the non-possible distribution in the incomplete algorithm.

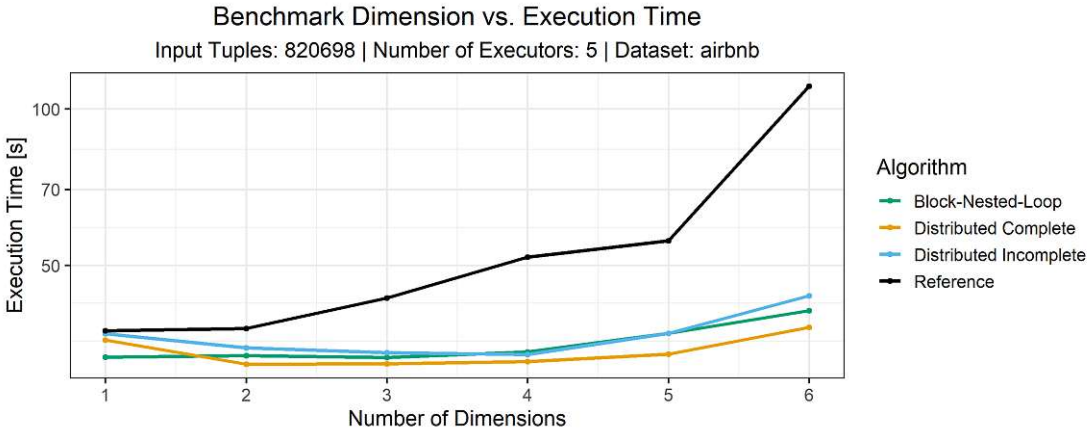

Figure 6.6: Number of dimensions vs. execution time on the complete Inside Airbnb dataset

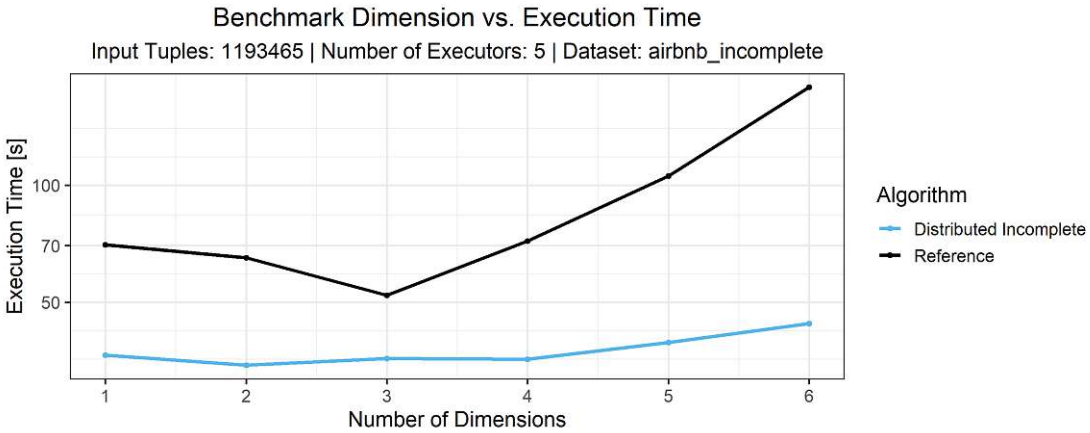

Figure 6.7: Number of dimensions vs. execution time on the complete Inside Airbnb dataset

Next, we look at how the number of executors affects the execution time in our settings. The results can be found in Figure 6.8 and Figure 6.9.

Here, it is most notable that 3 executors seems to be a sweet spot in the complete dataset. The runtime is the lowest for all algorithms except for the reference query.

In the incomplete dataset, there is a clearer direct correlation between the number of nodes and the execution time. We observe that the higher the number of nodes the lower the execution time for both the reference query and the distributed incomplete algorithm. The only exception to this is that 2 executors is slightly slower than using only a single executor. Such results can occur when the time needed to synchronize two different executors is higher than the time saved by using two executors to calculate the

local skyline. In case of the incomplete dataset, this heavily depends on the dataset itself since clustering occurs according to the `null` values which might be heavily skewed.

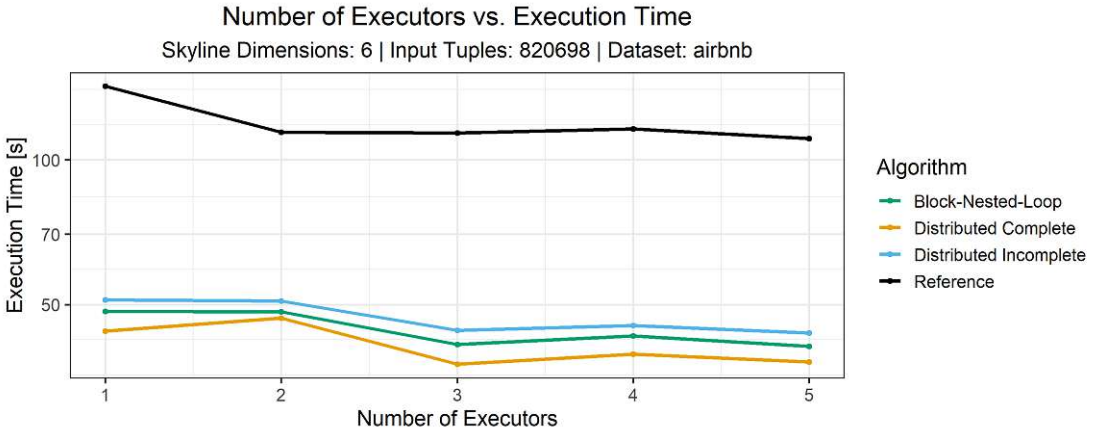

Figure 6.8: Number of nodes vs. execution time on the complete Inside Airbnb dataset

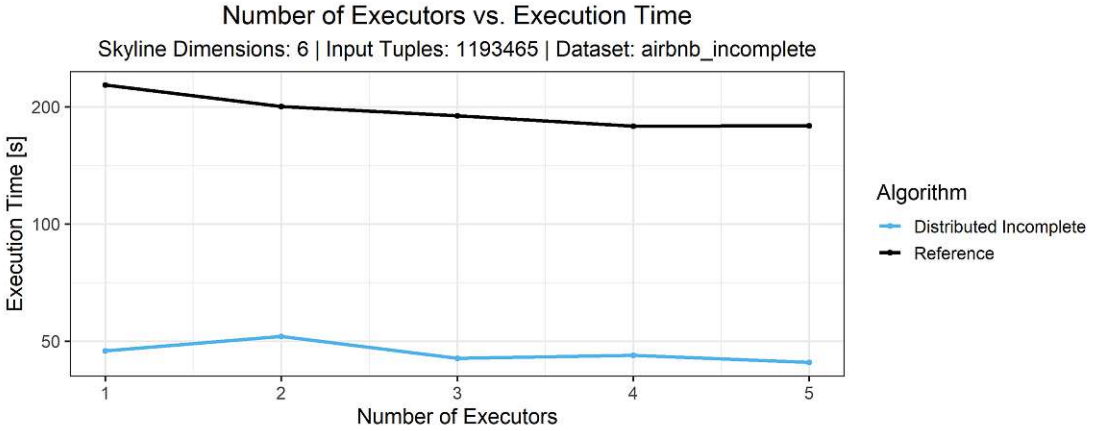

Figure 6.9: Number of nodes vs. execution time on the incomplete Inside Airbnb dataset

To round off our analysis of the real-world dataset, we look at how the execution time is affected by the size of the dataset in a clustered environment. The results can be found in Figure 6.10. We note that the data comes from different datasets (NBA, Fuel Economy, COIL 2000, and Inside Airbnb) which may introduce some inaccuracies.

The non-distributed algorithms outperform the distributed ones for the smaller datasets while for the bigger ones, the advantages of the distributed approaches become much more apparent. This effect is most clear for the distributed complete approach here which is outperformed by block-nested loop for the smaller datasets but trumps when it comes to the biggest dataset.

Figure 6.10: Dataset size vs. execution time on the complete real-world datasets

6.3.4.2.2 Synthetic Dataset

We now look at how the algorithms perform when using the bigger synthetic dataset. The main difference between these tests is that for the synthetic dataset we have a more predictable distribution of data and can more easily influence the size of each dataset. Due to the size of the benchmarks, we set a limit of 1 hour (3600 seconds) total execution time before the query is to be terminated.

The reference solutions are partially omitted in these plots since they either do not finish within 3600 seconds or crash without returning a result for most of the queries.

We first, look at how the number of dimensions influences the execution time. The results can be found in Figure 6.11 and Figure 6.12. For the latter, we choose the smaller dataset with "only" 1 million tuples since the reference solutions do not time out here.

For the complete dataset found in Figure 6.11, there is a clear correlation between the number of dimensions and the execution time. Given a high number of dimensions, the distributed complete algorithm outperforms the others by a big margin. All our algorithms outperform the reference solution for every number of dimensions. The correlation is not as clear for the reference solution which times out for the 1-dimensional skyline and performs badly for the 2-dimensional skyline.

This correlation does not appear for the incomplete dataset in Figure 6.12. Here, the execution time for the reference solution reaches its peak at 2 dimensions. It is noticeable that the same peak can also be found in the data of the distributed incomplete skyline algorithm. The likely explanation for this behavior is that the resulting skyline is large. This requires more checks the underlying block-nested-loop of the distributed incomplete algorithm which degrades the performance.

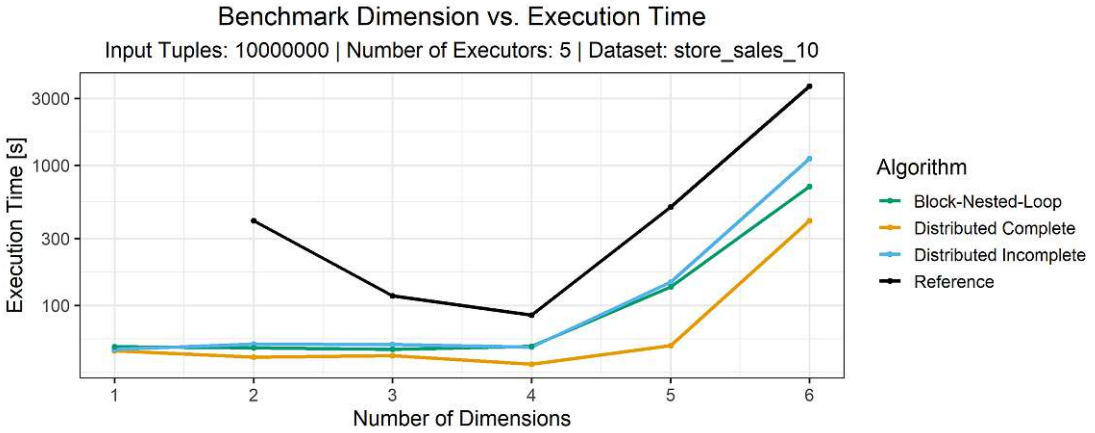

Figure 6.11: Number of dimensions vs. execution time on the complete store_sales dataset with 10 million tuples

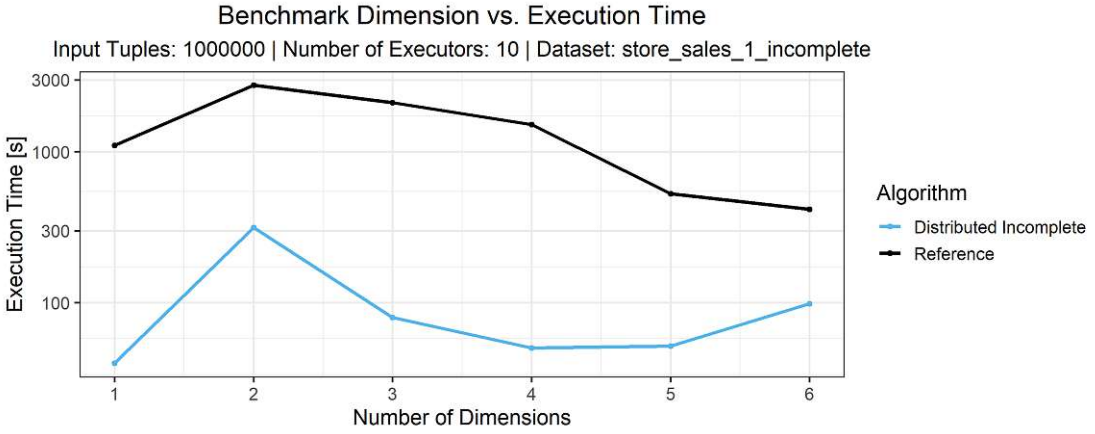

Figure 6.12: Number of dimensions vs. execution time on the incomplete store_sales dataset with 1 million tuples

Next, we observe the effects of the number of nodes on the execution time. For this, we deliberately chose the biggest datasets since the effects are more consistent and noticeable. The results can be found in Figure 6.13 and Figure 6.14.

For the distributed incomplete algorithm in Figure 6.13, it is most noticeable that a higher number of executors does not always increase performance. This is often caused by additional synchronization efforts. A similar effect can be observed for the distributed complete algorithm. Using 5 executors increases the performance significantly while there is a slight decrease for 10 executors. Here, this is due to the fact that the input sets to the local skylines are smaller which results in less eliminated tuples during the local skyline phase. Due to the increased input size, the global skyline computation then takes longer. It follows that there is a sweet spot for number of executors for the distributed algorithm

which depends heavily on the size of the dataset and the distribution of the data. For the reference solution, there is no improvement given a higher number of executors.

Given the distributed incomplete algorithm on the incomplete dataset (Figure 6.14), the execution time is much more erratic. This is likely due to the massive influence of the data itself on the partitioning. While a higher number of executors can bring performance increases, they depend heavily on whether the distribution is able to use those executors without suffering from the additional synchronization penalties and data skew in the local skyline phase. There are no data points for the reference solution since it times out for every number of executors.

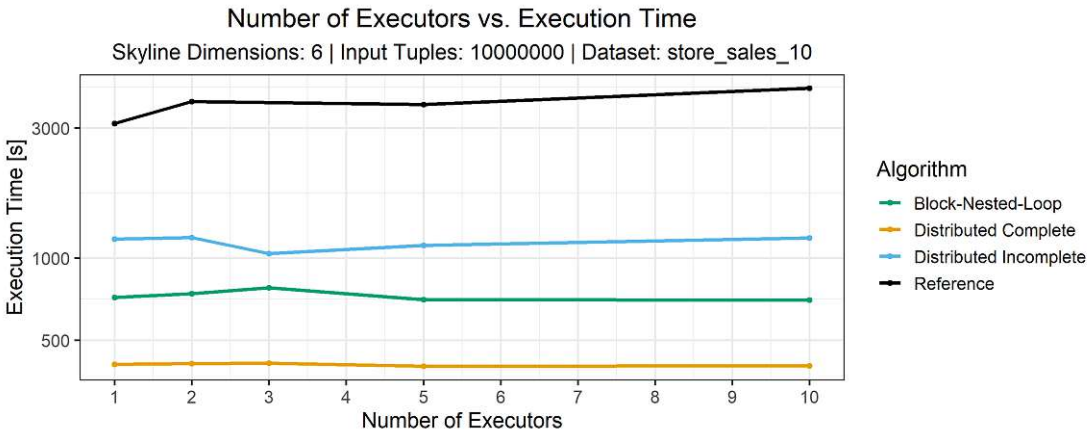

Figure 6.13: Number of executors vs. execution time on the complete store_sales dataset with 10 million tuples

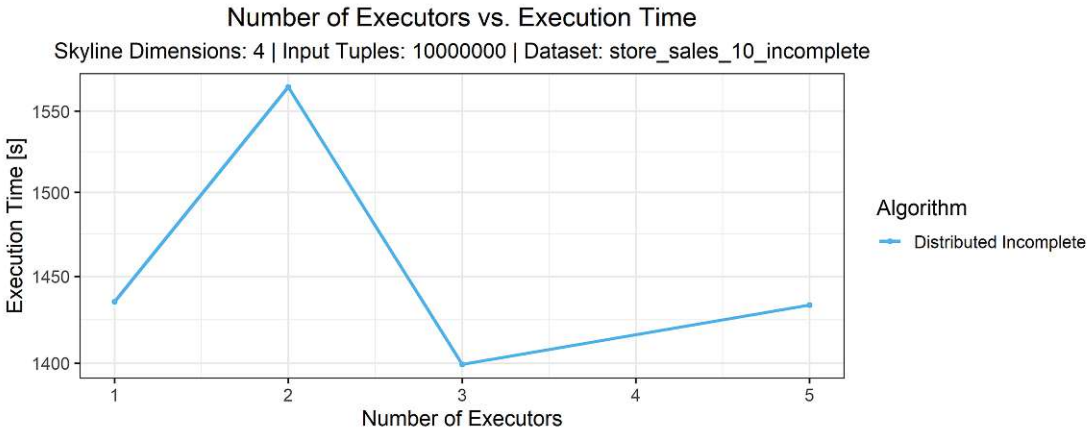

Figure 6.14: Number of executors vs. execution time on the incomplete store_sales dataset with 10 million tuples

We also look at how the size of the dataset affects the execution time in Figure 6.15. It is immediately noticeable that a bigger size of the dataset correlates with a higher execution time across all our algorithms. The distributed algorithm can cope with increased size better than the block-nested-loop and incomplete distributed algorithm. It can be seen across all dataset sizes that the reference solution performs worse than our algorithms. It follows that the main strengths of the distributed approach become more apparent the bigger the size of the dataset is.

Figure 6.15: Dataset size vs. execution time on the complete store_sales dataset with 10 executors

We now take a look at the memory usage and which factors have an influence on it. As part of this thesis, we check the influence of the number of dimensions and the number of nodes. The results can be found in Figure 6.16 and Figure 6.17.

In case of the complete dataset (Figure 6.16), the trend is that a higher number of dimensions also leads to higher memory consumption for all algorithms. The exact memory consumption is subject to a lot of jitter such that a lower number of dimensions may still consume more memory than expected. It is notable that the reference solution consumes the most memory. For the 2-dimensional skyline, the reference solution also consumes significantly more memory than for 3 dimensions. There is no data point available for the 1-dimensional reference query since it times out.

No such observation can be made for the incomplete dataset (Figure 6.17). The memory consumption does not seem to directly correlate with the number of dimensions for both the distributed incomplete algorithm and the reference query.

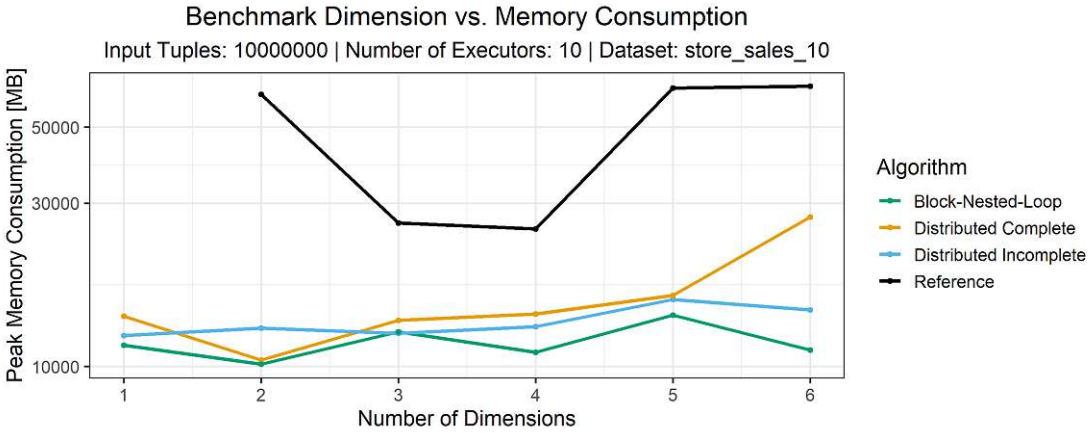

Figure 6.16: Number of dimensions vs. peak memory usage on the complete store_sales dataset with 10 million tuples

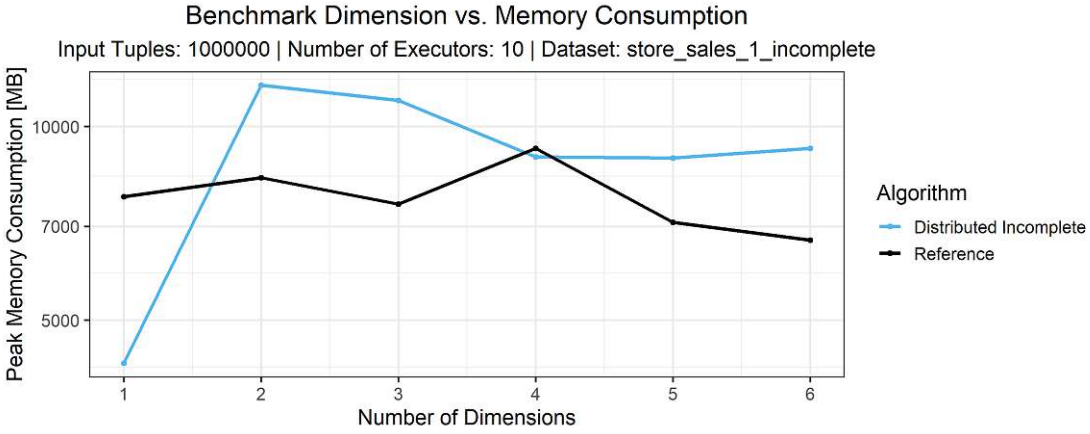

Figure 6.17: Number of dimensions vs. peak memory usage on the incomplete store_sales dataset with 1 million tuples

In Figure 6.18, we can see that the memory consumption does scale almost linearly with the number of executors. Here, the distributed approaches have a higher (peak) memory consumption since the data needs to be distributed more. The reference solution consumes even more memory than the distributed approaches.

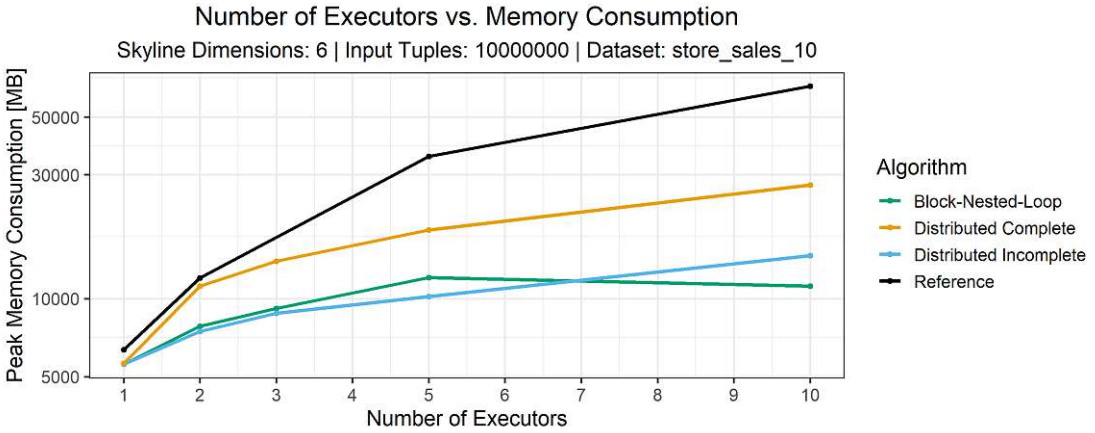

Figure 6.18: Number of executors vs. peak memory usage on the complete store_sales dataset with 10 million tuples

6.3.4.3 Conclusion

We can conclude from the results in this section that our implementation of skyline queries performs well even with big datasets.

Our distributed algorithms can make use of the distributed nature of Spark and use multiple executors and cores. There is, however, a "sweet-spot" for how many executors make the most sense for each skyline query. After that point, there are not enough tuples eliminated during the local skyline computation which means that the input size for the global skyline computation step grows. This then subsequently increases the time needed for the global computation skyline computation steps which delays the execution of the entire query.

The distributed incomplete algorithm should only be used if the underlying input data is actually incomplete or if it is unknown whether the data is incomplete. It is outperformed by both the complete distributed approach and the block-nested-loop algorithm when the dataset is complete.

The total memory consumption of the queries correlates with the number of executors used. There is also a correlation between the number of dimensions and the memory consumed but it is much less clear and more jittery.

CHAPTER 7

Conclusion and Future Work

We now recapitulate our achievements in this chapter. To round the thesis off, we also identify directions for further improvements.

Skyline queries are a useful tool for data scientists and programs like recommendation engines. However, they are currently not supported by Apache Spark. Our goal has been to find a way to combine skyline queries with the distributed environment offered by Apache Spark. Combining them brings many advantages since Spark is both widely used and suitable for the large datasets which are typical for skyline queries in production environments. To achieve our goals, we have selected Spark SQL as a suitable basis since we can write queries using a SQL-like query language or an easy-to-use API. Our main goal is the direct and proper integration of skyline queries into Spark SQL.

The source code is a modified version of Spark and available as open source on GitHub at https://github.com/Lukas-Grasmann/Spark_3.1.2_Skyline/. We also provide utilities for setup, tests, and benchmarks as open source at https://github.com/Lukas-Grasmann/Spark_Skyline_Utilities/.

7.1 What has been achieved?

We have successfully integrated skyline queries into Apache Spark SQL. This allows us to conveniently write skyline queries and execute them such that the skylines of potentially large datasets can be retrieved efficiently.

Queries can now be written as query strings or by using Spark's DataFrame/DataSet API. We have achieved this by extending the syntax of Spark SQL query strings and by adding new methods to the DataFrame/DataSet API. To allow Spark to properly process the skyline queries, we have added corresponding nodes to the execution plans.

We have successfully implemented three different algorithms which all have a specific strength.

- The **block-nested-loop** algorithm is simple and effective on smaller datasets. It provides a good basis for further algorithms.
- The **complete distributed** algorithm is based on MapReduce and offers a great performance improvement over the block-nested-loop algorithm for bigger datasets. With this algorithm, we successfully take advantage of the distributed processing nature of Spark by splitting parts of the skyline computations into independently manageable parts.
- The **incomplete distributed** algorithm allows us to compute the skylines of incomplete datasets. This greatly expands the envelope of datasets which are suitable as input for our integration. It is the best algorithm for datasets which are incomplete or where it is not known whether they are incomplete or complete.

Given these algorithms, we are able to compute the skylines of a wide range of datasets. As opposed to many other systems, we are not limited to either complete or incomplete datasets.

We have taken advantage of Spark SQL's own Catalyst optimizer by adding additional optimizations rules to improve the performance of our skyline processing. These optimizations mostly handle special cases such as empty skylines or skylines which can be pushed through joins. They help with errors by the user when writing skyline queries by removing empty skylines and redundant skyline dimensions.

Our solution has been extensively tested by both unit and system tests to ensure that it is working correctly. We have used datasets from the real world as well as synthetic datasets to test the performance of our system by running benchmarks on both a standalone local machine and in a distributed cluster environment.

In these benchmarks, it is notable that our solution outperforms the reference queries by big margins. Using the more advanced distributed complete algorithm instead of a block-nested-loop has proven to be the best course of action since we can reduce the execution time significantly especially for larger datasets and higher numbers of dimensions. For incomplete datasets, only the distributed incomplete algorithm is suitable. It outperforms the corresponding reference queries.

We conclude that the productive and efficient integration into Spark is possible and has successfully been achieved in this thesis. Using a distributed approach based on MapReduce, we can make use of the distributed processing capabilities of Spark to quickly and effectively compute skylines. A modification of said algorithm enables us to retrieve skylines of incomplete datasets. Our solutions are much faster than "plain" SQL queries as we have shown through benchmarks. It is also compatible with a wide range of database systems by using Apache Spark. Writing queries is significantly more convenient now since we can use the query string syntax and DataFrame/DataSet API we have implemented.

7.2 What comes next?

While our integration of skyline queries into Spark already works well and clearly outperforms the reference queries, there are still areas which can be improved beyond the scope of this thesis. We will now look at areas where such improvements are possible.

One possibility to improve the integration of skyline queries into Spark is to make them available in other programming languages. This, for example, includes the integration into the mathematical programming language R which, aside from Python, is a prime candidate for writing programs as a data analyst. In principle, such integrations are very similar to the ones already introduced in this thesis.

We have implemented an array of optimizations using the Catalyst optimizer. Despite this, there are still some optimizations which can be implemented using additional rules or modified algorithms. This includes, for example, pushing skylines into joins and some special cases of pushing skylines through joins.

There are algorithms not based on block-nested-loop which can be integrated into Spark SQL. This includes the divide-and-conquer approaches and index-based algorithms. The latter suffer from the lack of support for indexes in Spark and would require a very significant amount of preliminary work to be done.

Aside from the fundamentally different algorithms, the block-nested-loop can be further improved upon. This can be done by using more advanced data structures such as self-organizing buffers and caches. The main purpose of this is to reduce the number of comparisons necessary to determine dominance by having “killer tuples” at the beginning of the window.

Last but not least, there are further improvements to the distributed algorithms and partitioning schemes. In this thesis, we use the default Spark partitioning scheme which provides good results but can be improved upon. These improvements include grid-based partitioning schemes as well as angle-based partitioning schemes.

The former has the additional advantage that it allows us to use more advanced distributed algorithms that include advanced features like removing entire grid cells due to dominance. This is significantly harder to do than the algorithms implemented in this thesis since we first need to find the cells and then handle metadata such as best and worst point of each cell.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ANT] Antlr. Accessed on: 15.07.2021. URL: https://www.antlr.org/.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015. doi:10.1145/2723372.2742797.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 421–430. IEEE Computer Society, 2001. doi:10.1109/ICDE.2001.914855.

[CK97] Michael J. Carey and Donald Kossmann. On saying "enough already!" in SQL. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 219–230. ACM Press, 1997. doi:10.1145/253260.253302.

[Clu] Spark - cluster mode overview. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/cluster-overview.html.

[CLX+08] Bin Cui, Hua Lu, Quanqing Xu, Lijiang Chen, Yafei Dai, and Yongluan Zhou. Parallel distributed processing of constrained skyline queries by filtering. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 546–555. IEEE Computer Society, 2008. doi:10.1109/ICDE.2008.4497463.

[COI] Coil 2000. Accessed on: 19.01.2022. URL: https://archive.ics.uci.edu/ml/datasets/Insurance+Company+Benchmark+(COIL+2000).

[DCGN21] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. Dsb: A decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.*, 14(13):3376–3388, sep 2021. doi:10.14778/3484224.3484234.

[DS07] Evangelos Dellis and Bernhard Seeger. Efficient computation of reverse skyline queries. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 291–302. ACM, 2007. URL: http://www.vldb.org/conf/2007/papers/research/p291-dellis.pdf.

[DSB] Dsb on github. Accessed on: 19.01.2022. URL: https://github.com/microsoft/dsb.

[DSt] Apache spark dstreams. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[Ede09] Hannes Eder. On extending postgresql with the skyline operator. Master's thesis, Fakultät für Informatik der Technischen Universität Wien, 1 2009.

[Fag99] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999. doi:10.1006/jcss.1998.1600.

[Fue] Fuel economy. Accessed on: 19.01.2022. URL: https://www.fueleconomy.gov/feg/download.shtml.

[GAT19] Yonis Gulzar, Ali Amer Alwan, and Sherzod Turaev. Optimizing skyline query processing in incomplete data. *IEEE Access*, 7:178121–178138, 2019. doi:10.1109/ACCESS.2019.2958202.

[Gra] Apache spark graphx. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/graphx-programming-guide.html.

[GWM+19] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 467–484. ACM, 2019. doi:10.1145/3299869.3324959.

[HV12] Katja Hose and Akrivi Vlachou. A survey of skyline processing in highly distributed environments. *VLDB J.*, 21(3):359–384, 2012. doi:10.1007/s00778-011-0246-6.

[Ins] Inside airbnb. Accessed on: 19.01.2022. URL: http://insideairbnb.com/get-the-data.html.

[KK18] Junsu Kim and Myoung Ho Kim. An efficient parallel processing method for skyline queries in mapreduce. *J. Supercomput.*, 74(2):886–935, 2018. doi:10.1007/s11227-017-2171-y.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark - Lightning-Fast Data Analysis*. O'Reilly Media, Inc., 1005 Gravenstein Highway North Sebastopol, CA 95472 USA, 2015.

[KLP75] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975. doi:10.1145/321906.321910.

[KT17] Christos Kalyvas and Theodoros Tzouramanis. A survey of skyline query processing. *CoRR*, abs/1704.01788, 2017. URL: http://arxiv.org/abs/1704.01788, arXiv:1704.01788.

[KW17] Holden Karau and Rachel Warren. *High Performance Spark*. O'Reilly Media, Inc., 1005 Gravenstein Highway North Sebastopol, CA 95472 USA, 2017.

[LZLL07] Ken C. K. Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. Approaching the skyline in Z order. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 279–290. ACM, 2007. URL: http://www.vldb.org/conf/2007/papers/research/p279-lee.pdf.

[MLl] Apache mlib. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/ml-guide.html.

[NBA] Nba stats. Accessed on: 19.01.2022. URL: https://www.nba.com/stats/.

[Pap18] Ioanna Papanikolaou. Distributed algorithms for skyline computation using apache spark. Master's thesis, 2018. Accessed on: 23.01.2022. URL: https://repository.ihu.edu.gr/xmlui/handle/11544/29524.

[Py4] Py4j - a bridge between python and java. Accessed on: 07.09.2021. URL: https://www.py4j.org/.

[PyS] Apache pyspark. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/api/python/getting_started/index.html.

[RMF+00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the ub-tree into a database system kernel. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 263–272. Morgan Kaufmann, 2000. URL: http://www.vldb.org/conf/2000/P263.pdf.

[Sha] Shark, spark sql, hive on spark, and the future of sql on apache spark. Accessed on: 12.07.2021. URL: https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html.

[Sky] Skyspark. Accessed on: 10.11.2020. URL: https://github.com/AkiKanellis/skyspark.

[Spaa] Apache spark documentation. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/.

[Spab] Apache spark java documentation. Accessed on: 08.05.2021. URL: https://spark.apache.org/docs/latest/api/java/index.html.

[Spac] Apache spark scala documentation. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/api/scala/org/apache/spark/index.html.

[Spad] Apache spark sparkr. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/sparkr.html.

[Spae] Apache spark sql ansi compatibility. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/sql-ref-ansi-compliance.html.

[Spaf] Apache spark sql ansi compatibility. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/SparkSessionExtensions.html.

[Spag] Apache spark sql documentation. Accessed on: 12.07.2021. URL: https://spark.apache.org/docs/latest/api/sql/index.html.

[Spah] Apache spark strutured streaming. Accessed on: 14.06.2021. URL: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[Spai] Apache spark website. Accessed on: 12.07.2021. URL: https://spark.apache.org/.

[Spaj] Spark release 3.0.0. Accessed on: 15.06.2021. URL: https://spark.apache.org/releases/spark-release-3-0-0.html.

[Spak] Spark sql & dataframes. Accessed on: 12.07.2021. URL: https://spark.apache.org/sql/.

[Tid] Tidyverse. Accessed on: 12.07.2021. URL: https://www.tidyverse.org/.

[TYA+19] Mingjie Tang, Yongyang Yu, Walid G. Aref, Qutaibah M. Malluhi, and Mourad Ouzzani. Efficient parallel skyline query processing for high-dimensional data. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 2113–2114. IEEE, 2019. doi:10.1109/ICDE.2019.00251.

[VDK08] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 227–238. ACM, 2008. doi:10.1145/1376616.1376642.

[WXG+20] Jin Wang, Guorui Xiao, Jiaqi Gu, Jiacheng Wu, and Carlo Zaniolo. RASQL: A powerful language and its system for big data applications. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2673–2676. ACM, 2020. doi:10.1145/3318464.3384677.