Integration of Skyline Queries into Spark SQL

Lukas Grasmann
lukas.grasmann@tuwien.ac.at
TU Wien
Vienna, Austria

Reinhard Pichler
reinhard.pichler@tuwien.ac.at
TU Wien
Vienna, Austria

Alexander Selzer
alexander.selzer@tuwien.ac.at
TU Wien
Vienna, Austria

ABSTRACT

Skyline queries are frequently used in data analytics and multi-criteria decision support applications to filter relevant information from big amounts of data. Apache Spark is a popular framework for processing big, distributed data. The framework even provides a convenient SQL-like interface via the Spark SQL module. However, skyline queries are not natively supported and require tedious rewriting to fit the SQL standard or Spark's SQL-like language.

The goal of our work is to fill this gap. We thus provide a full-fledged integration of the skyline operator into Spark SQL. This allows for a simple and easy to use syntax to input skyline queries. Moreover, our empirical results show that this integrated solution by far outperforms a solution based on rewriting into standard SQL.

1 INTRODUCTION

Skyline queries are an important tool in data analytics and decision making to find *interesting* points in a multi-dimensional dataset. Given a set P of data points, we can compute the skyline (also called Pareto front) using the *skyline operator* [5]. Intuitively, a data point $p \in P$ belongs to the skyline if it is not *dominated* by any other point. For a given set of dimensions, a data point q *dominates* p (denoted $q \prec p$) if q is better in at least one dimension while being at least as good in every other dimension. More formally, the skyline S is obtained as $S = \{p \in P \mid \nexists q \in P : q \prec p\}$.
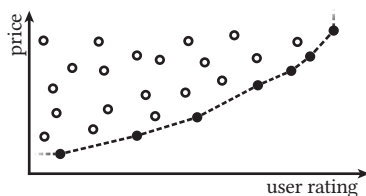

Figure 1: Example skyline for hotels

A common example for skyline queries in the literature can be found in Figure 1. Suppose that we want to find the perfect hotel for our holiday stay. Each hotel has multiple properties that express its quality such as the price per night or the distance to the beach among others. In this example, we limit ourselves to the price per night (y-axis) and the user rating (x-axis) for simplicity. Such attributes relevant to the skyline are called *skyline dimensions*. A hotel dominates another hotel if it is strictly better in at least one skyline dimension and not worse in all others. For the price per night, minimization is desirable while we want the rating to be as high as possible (maximization). Figure 1 depicts the skyline of the hotels according to these two dimensions. In

this simple (two-dimensional) example, it is immediately visible that for each hotel not part of the skyline there exists a "better" (dominating) alternative. In practice, things get more complex as skyline queries usually handle a larger number of skyline dimensions.

Skyline queries have a wide range of applications, including recommender systems to propose suitable items to the user [5] and location-based systems to find "the best" routes [23, 28]. Skyline queries have also been used to improve the quality of (web) services [2] and they have applications in privacy [7], authentication [33], and DNA searching [8]. Many further use cases for skyline queries are given in these surveys: [9, 22, 24]. Skyline queries are particularly useful when dealing with big datasets and filtering the relevant data items. Indeed, in data science and data analytics, we are typically confronted with big data, which is stored in an optimized and distributed manner. We thus need a processing engine that is capable of efficiently operating on such data.

Apache Spark [45] is a unified framework that is specifically designed for distributed processing of potentially big data and has gained huge popularity in recent years. It works for a wide range of applications, and it can interoperate with a great variety of data sources. Conceptually, it is similar to the MapReduce framework, where instead of map-combine-reduce nodes, there are individual nodes in an execution plan for each processing step. Spark also provides a multitude of modular extensions such as MLlib, PySpark, SparkR, and others, built on top of its core functionality. Another important extension is Spark SQL, which provides relational query functionality and comes with the powerful *Catalyst* optimizer.

Hence, Apache Spark would be the perfect fit for executing skyline queries over big, distributed data. Despite this, to date, the skyline operator has not been integrated into Spark SQL. Nevertheless, it is possible to formulate skyline queries using "plain" SQL. Listing 2 shows the "plain" skyline query for our hotel example.

Listing 1: Hotel skyline query in plain SQL [5, 15]

```
1 SELECT price, user_rating FROM hotels AS o WHERE
    NOT EXISTS(
2     SELECT * FROM hotels AS i WHERE
3         i.price ≤ o.price
4         AND i.user_rating ≥ o.user_rating
5         AND (
6             i.price < o.price
7             OR i.user_rating < o.user_rating
8         )
9 );
```

There are several drawbacks to such a formulation of skyline queries in plain SQL. First of all, as more skyline dimensions are used, the SQL code gets more and more cumbersome. Hence, such

an approach is error-prone, and readability and maintainability will inevitably be lost. Moreover, most SQL engines (this also applies to Spark SQL) are not optimized for this kind of nested SELECTs. Hence, also performance is of course an issue. Therefore, a better solution is called for. This is precisely the goal of our work:

- We aim at an integration of skyline queries into Spark SQL with a simple, easy to use syntax, e.g., the skyline query for our hotel example should look as depicted in Listing 2.
- We want to make use of existing query optimizations provided by the Catalyst optimizer of Spark SQL and extend the optimizer to also cover skyline queries.
- At the same time, we want to make sure that our integration of the skyline operator has no negative effect on the optimization and execution of other queries.
- Maintainability of the new code is an issue. We aim at a modular implementation, making use of existing functionality and making future enhancements as easy as possible.
- Spark and Spark SQL support many different data sources. The integration of skyline queries should work independently of the data source that is being used.

Listing 2: Hotel skyline query in extended SQL [5, 15]

```
1 SELECT price, user_rating FROM hotels SKYLINE OF
    price MIN, user_rating MAX;
```

Our contribution. The main result of this work is a full-fledged integration of skyline queries. This includes the following:

- Virtually every single component involved in the query optimization and execution process of Spark SQL (see Figure 2) had to be extended: the parser, analyzer, optimizer, etc.
- While a substantial portion of new code had to be provided, at the same time, we took care to make use of existing code as much as possible and to keep the extensions modular for easy maintenance and future enhancements.
- For the optimization of skyline queries, the specifics of distributed query processing had to be taken into account. Apart from incorporating new rules into the Catalyst optimizer of Spark SQL, we have extended the skyline query syntax compared with previous proposals [5, 15] to allow the user to control further skyline-specific optimization.
- We have carried out an extensive empirical evaluation which shows that the integrated version of skyline queries by far outperforms the rewritten version in the style of Listing 1.

Structure of the paper. In Section 2, we discuss relevant related work. The syntax and semantics of skyline queries are introduced in Section 3. A brief introduction to Apache Spark and Spark SQL is given in Section 4. In Section 5, we describe the main ideas of our integration of the skyline operator into Spark SQL. We report on the results of our empirical evaluation in Section 6. In Section 7, we conclude and identify some lines of future work.

The source code of our implementation is available here: https://github.com/Lukas-Grasmann/skyline-queries-spark. Further material, comprised of the binaries of our implementation, all input data, and queries from our empirical evaluation is provided at [20].

2 RELATED WORK

Since the original publication of the skyline operator by Börzsönyi, Kossmann, and Stocker [5], various algorithms and approaches to skyline queries have been proposed. In this section, we give an overview of the relevant related work. We limit ourselves to, broadly, the "original" definition of skyline queries and exclude related query types like reverse skyline queries [13].

The main cost factor of skyline computation is the time spent on dominance testing, which in turn depends on how many dominance tests between tuples need to be performed. A simple, straightforward algorithm runs in quadratic time w.r.t. the size of the data. It checks for each pair of tuples whether one of them is dominated and, therefore, not part of the resulting skyline. This basic idea of pairwise dominance tests is realized in the Block-Nested-Loop skyline algorithm [5], which performs well especially when the resulting output is not too big, and the input dataset fits in memory.

In the most recent algorithms, there are two main approaches that are used to either decrease the total number of dominance checks or to increase the parallelism of these checks.

(1) Using a score function on the skyline dimensions and sorting the tuples w.r.t. to this score function [5, 34]. These approaches may require the entire dataset to be available on a single node, which is not ideal in a distributed environment.
(2) Partitioning (and distributing) the skyline computation by first independently computing the skyline of partitions (*local skylines*) before proceeding to the computation of the final skyline (*global skyline*) [12, 21, 26, 42]. This does not necessarily decrease the total number of dominance checks but speeds up the algorithm through parallelism.

Most of these approaches are centered around special data structures and indexes, which are provided by the respective database system. This makes them less suitable for Spark, which currently has no internal support for building and maintaining indexes. Such algorithms include Index [41] which uses a bitmap index to compute the skyline. Many algorithms of this class also make use of presorting, i.e., sorting the tuples from the dataset according to a monotone scoring function. Such scoring can, by definition, rule out certain dominance relationships between tuples such that fewer comparisons become necessary. This includes the algorithms SFS [10, 11], LESS [17, 18], SALSA [3, 4], and SDI [34].

A simple partition-based algorithm is the Divide-and-Conquer algorithm presented in the original skyline paper [5]. Its working principle is to recursively divide the data into partitions until the units are small enough such that the skyline can be computed efficiently. The resulting (local) skylines are then merged step by step until the entire recursion stack has been processed. Related approaches include the nearest-neighbor search [27] as well as other similar algorithms [35–37, 40, 46].

The algorithm BSkyTree [29, 30] supports both a sorting-based and a partition-based variant. These make use of an index tree-structure to increase performance. Alternatively, there also exist tree structures based on quad-trees [39]. Here, the data is partitioned dynamically and gradually into smaller and smaller "rectangular" chunks until the skylines can be computed efficiently.

There are also newer algorithms specifically targeted towards the MapReduce framework [26, 32, 42]. MapReduce approaches use *mappers* to generate partitions by assigning keys to each

tuple while the *reducers* are responsible for computing the actual skylines for all tuples with the same key. Each intermediate skyline for a partition is called a *local skyline*. This approach usually employs two or more rounds of map and reduce operations (*stages*) where each stage first (re-)partitions the data and then computes skylines. The final result is then derived by the last stage which generates only one partition and computes the *global skyline*.

The success of MapReduce approaches relies on keeping the intermediate results as small as possible and maximizing parallelism [26]. To achieve this, multiple different partitioning schemes exist such as *random*, *grid-based*, and *angle-based* partitioning [26]. Keeping the last local skylines before the global skyline computation small is especially desirable since the global skyline computation cannot be fully parallelized. Indeed, this is usually the bottleneck which slows the skyline computation down [26]. This approach is not suitable for our purposes since it requires data from different partitions to be passed along with meta-information about the partitions. Spark does not currently support such functionality and adding it would be beyond the scope of this work.

There are efforts to eliminate multiple data points at once and make the global skyline computation step distributed by partitioning the data. In the case of *grid-based-partitioning* it is possible to eliminate entire *cells* of data if they are dominated by another (non-empty) cell [42]. It is then also possible to cut down the number of data points which each tuple must be compared against in the global skyline since some cells can never dominate each other according to the basic properties of the skyline (if a tuple is better in at least one dimension then it cannot be dominated) [26, 42].

There have been two past efforts to implement skyline computation in Spark. The first one is the system *SkySpark* [25] which uses Spark as the data source and realizes the skyline computation on the level of RDD operations (see Section 4 for details on Spark and RDDs). A similar effort was made in the context of the thesis by Ioanna Papanikolaou [38], which uses the map-reduce functionality of RDDs in Spark to compute the skyline. Both systems thus realize skyline computation as standalone Spark programs rather than as part of Spark SQL, which would make, for instance, a direct performance comparison with our fully integrated solution difficult (even though it might still provide some interesting insights). However, the most important issue is that these implementations are *not ready to use*: SkySpark [25], with the last source code update in 2016, implements skyline computation for an old version of Spark. The code has been deprecated and archived by the author with a note that the code does not meet his standards anymore. For the system reported in [38], there is no runnable code provided since it is only available as code snippets in the PDF document.

As will be detailed in Section 5, we have chosen the Block-Nested-Loop skyline algorithm [5] for our integration into Spark SQL due to its simplicity. By the modular structure of our implementation, replacing this algorithm by more sophisticated ones in the future should not be too difficult. We will come back to suggestions for future work in Section 7. Partitioning also plays an important role in our distributed approach (see Section 5 for limits of distributed processing in the case of incomplete data though). However, to avoid unnecessary communication cost, we refrain from overriding Spark's partitioning mechanism.

3 SKYLINE QUERIES

In this section, we take a deeper look into the syntax and semantics of skyline queries. Skyline queries can be expressed in a simple extension of SELECT-FROM-WHERE queries in SQL. This syntax was proposed together with the original skyline operator in [5] and can be found in Listing 3. The m skyline dimensions out of n total dimensions are denoted as d_1 through d_m (where $m \leq n$). The DISTINCT keyword defines that we only return a single tuple if there are tuples with the same values in the skyline dimensions. If there are multiple possibilities, the exact tuple that is returned is chosen arbitrarily. The keyword COMPLETE is an extension compared to [5] that we have introduced. It allows the user to inform the system that the dataset is complete in the sense that no null occurs in the skyline dimensions. The system can, therefore, safely choose the *complete* skyline algorithm, which is more efficient than the *incomplete* one. This will be discussed in more detail in Section 5.

Listing 3: Syntax of skyline queries in SQL [5]

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

In Section 1, we have already introduced the dominance relationship and the skyline itself in a semi-formal way. Below, we provide formal definitions of both concepts.

Definition 3.1 (Dominance relationship between tuples [5, 26]). Given a set of tuples R, two n-ary tuples $r, s \in R$ and a set of skyline dimensions D which always consists of four (potentially empty) disjoint subsets $D_{min}, D_{max}, D_{diff}, D_{extra}$, we define r_i and s_i as the value of the tuple r and s in dimension $d_i \in D$ respectively. The subsets of the dimensions correspond to the skyline dimensions ($D_{min}, D_{max}, D_{diff}$) and "extra" (non-skyline) dimensions (D_{extra}). Then r *dominates* s ($r \prec s$) if and only if:

$$\left(\bigwedge_{d_i \in D_{min}} r_i \leq s_i \right) \wedge \left(\bigwedge_{d_i \in D_{max}} r_i \geq s_i \right) \wedge \left(\bigwedge_{d_i \in D_{diff}} r_i = s_i \right)$$

$$\wedge \left(\left(\bigvee_{d_i \in D_{min}} r_i < s_i \right) \vee \left(\bigvee_{d_i \in D_{max}} r_i > s_i \right) \right)$$

In words, the above definition means that a tuple $r \in R$ *dominates* another tuple $s \in R$ if and only if:

- The values in all DIFF dimensions are equal **and**
- r is *at least as good* in all MIN/MAX skyline dimensions **and**
- r is *strictly better* in at least one MIN/MAX skyline dimension

The dominance relationships are *transitive*, i.e., if a dominates b and b dominates c then a also dominates c.

Given this formal definition of the dominance relationship, we can now also formally define the skyline itself.

Definition 3.2 (Skyline [5]). Given a set of skyline dimensions D, let R be a set of tuples. The *skyline R* (denoted $SKY(R)$) is a set of tuples defined as follows: $SKY(R) := \{r \in R \mid \nexists s \in R : s \prec r\}$.

Note that the above definitions apply to *complete* datasets. For *incomplete* datasets (if null may occur in some skyline dimension), we need to slightly modify the definition of dominance in that the comparison of two tuples is always restricted to those

dimensions where both are not `null`. Hence, the above definition for the complete case is modified as follows [21]:

- The values in all `DIFF` dimensions where r and s are not `null` are equal **and**
- r is *at least as good* in all `MIN/MAX` skyline dimensions where r and s are not `null` **and**
- r is *strictly better* in at least one `MIN/MAX` skyline dimension where r and s are not `null`

Given the modified dominance relationship, we now run into the problem that for incomplete datasets the *transitivity* property of skyline dominance is lost and there may be *cyclic dominance relationships*. For instance, assume a dataset similar to the one given in [21] consisting of three tuples $a = (1, *, 10)$, $b = (3, 2, *), c = (*, 5, 3)$ where $*$ is a placeholder for a missing value. If all three dimensions are skyline dimensions, then, given the definition of dominance in incomplete datasets, it follows that $a \prec b$ since $1 < 3$ for the first dimension. Similarly, we have $b \prec c$ since $2 < 5$ on the second dimension. Lastly, we note that $a \nprec c$ since $10 \nless 3$ but $c \prec a$ since $3 < 10$ for the third dimension. Under the assumption of transitivity, it would follow from $a \prec b$ and $b \prec c$ that $a \prec c$ holds, which is not the case. In other words, for incomplete data, the transitivity property gets lost [21]. Since $a \prec b$, $b \prec c$, and $c \prec a$, the dominance relationship forms a cycle and can therefore be referred to as a *cyclic dominance relationship* [21]. Hence, when computing the skyline of a potentially incomplete dataset, prematurely deleting dominated tuples may lead to erroneous results. Indeed, this is the trap into which the skyline algorithm proposed in [21] fell. For details, see [19].

As already mentioned in Section 1, skyline queries can be formulated in plain SQL without the specialized skyline syntax from Listing 3. A general schema of rewriting skyline queries in plain SQL is given in Listing 4. A first version of this rewriting was informally described in [5]. The full details and the corresponding syntax used here were introduced in [15]: First, the outer query selects all tuples from some relation (which may itself be the result of a complex SELECT-statement) and then we use a subquery with `WHERE NOT EXISTS` to eliminate all dominated tuples.

Listing 4: Translated skyline query in plain SQL [5, 15]

```
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.a₁ ≤ o.a₁ AND ... AND i.aⱼ ≤ o.aⱼ
4          AND i.aⱼ₊₁ ≥ o.aⱼ₊₁ AND ... AND i.aₖ ≥ o.aₖ
5          AND i.aₖ₊₁ = o.aₖ₊₁ AND ... AND i.aₘ = o.aₘ
6          AND (
7              i.a₁ < o.a₁ OR ... OR i.aⱼ < o.aⱼ
8              OR i.aⱼ₊₁ > o.aⱼ₊₁ OR ... OR i.aₖ > o.aₖ
9          )
10 );
```

---

4 APACHE SPARK

Apache Spark is a unified and distributed framework for processing large volumes of data. The framework can handle many different data sources. Through various specialized modules on top of its core module, it supports a wide range of processing workloads. The central data structure of the Spark Core are *resilient distributed datasets* (RDDs). An RDD is a collection of elements distributed over the nodes of the cluster. Working with Spark comes down to defining transformations and actions on RDDs, while data distribution, parallel execution, synchronization, fault-tolerance, etc. are taken care of by the system and are largely hidden from the user.

The focus of our work lies on the Spark SQL module, which extends the Spark Core by providing an SQL interface. Queries can be formulated either via query strings or using an API. This API is based on specialized data structures called *DataFrame* and *DataSet*. They are (meanwhile) closely related in that every DataFrame is also a DataSet, with the only difference being that DataSets are strongly typed while DataFrames are not. The processing and execution of input queries follows a well-defined schema with multiple steps that are depicted in Figure 2.

First, the queries are formulated either via SQL query strings or via APIs. In the case of query strings, it is necessary to parse them first. Both methods produce a *logical* execution plan that contains the information which operations the plan consists of. For instance, filtering is represented as a specific node in the plan and can be used for both `WHERE` and `HAVING` clauses in the query. The references to tables or columns are not yet assigned to actual objects in the database in this step. To solve this, the `Analyzer` takes each identifier and translates it using the `Catalog`. The result returned by the `Analyzer` is the *resolved* logical plan where all placeholders have been replaced by actual references to objects in the database.

The *Catalyst Optimizer* is a crucial part for the success of Spark SQL. It is a powerful rule-based optimizer that can be extended to include new rules for specific (newly introduced) nodes. Optimizations are applied to the resolved logical plan.

To actually execute the query, the logical execution plan must first be translated into a *physical* execution plan, which contains information about how the various operations are to be realized. For example, Spark provides different join implementations, which are each represented by different nodes in the physical plan and one of them is selected during the physical planning phase. There may be more than one physical plan generated during this step. Based on the physical plan(s), one specific plan must be selected, and *Code Generation* is carried out according to the chosen physical plan. In Spark, the generated code is a computation based on RDDs.

It will turn out that, for a full-fledged integration of skyline queries into Spark SQL, virtually all of the components of query processing and execution mentioned above have to be extended or modified. This will be the topic of the next section.

5 INTEGRATING SKYLINES INTO APACHE SPARK

Below, we give an overview of the required modifications and extensions of each component of the query processing flow depicted in Figure 2. An important aspect of integrating skyline queries into Spark SQL is the algorithm selection for the actual computation of the skyline. Here, we will discuss in some detail the challenges arising from potentially incomplete data and our solution.

5.1 Skyline Query Syntax in Spark SQL

The parser is the first step of processing a Spark SQL query. It uses the ANTLR parser grammar generator to generate a lexer and subsequent parser which take query strings as input to obtain
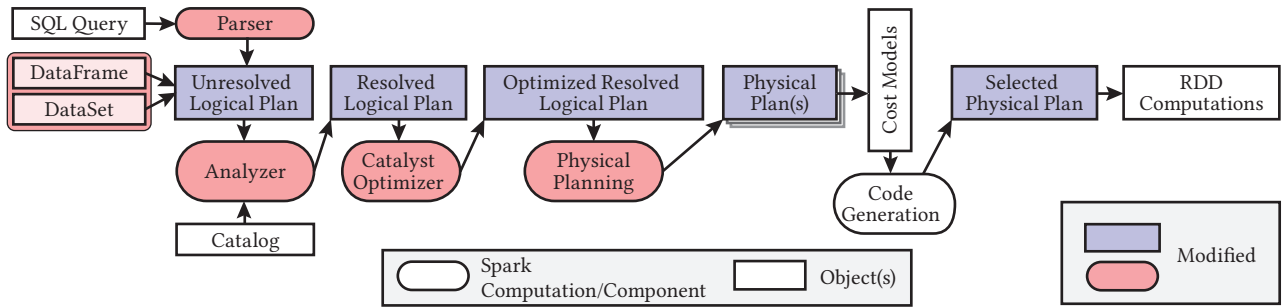
Figure 2: Apache Spark SQL Query Processing and Execution (adapted from [44])

nodes in a logical plan. We modify both parts to extend Spark's SQL-like syntax such that it also includes skyline queries.

Listing 5: ANTLR grammar for skyline queries

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

The grammar corresponding to the syntax introduced in Listing 3 is shown in Listing 5. Given a "regular" SELECT statement, note that a skyline always comes after the HAVING clause (if any) but before any ORDER BY clause. Each skyline clause consists of the SKYLINE OF keyword (line 2), an optional DISTINCT (line 3) and COMPLETE (line 4), as well as an arbitrary number of skyline dimensions (line 5 and 8 - 9) where the "type" MIN/MAX/DIFF is given separately for each dimension (line 9). The ordering of skyline dimensions in this syntax has no bearing on the outcome of the query (except for potential ordering). It may, however, have a slight effect on the performance of dominance checks since it determines the order in which the skyline dimensions of two tuples are compared.

5.2 Extending the Logical Plan

Each node in the logical plan stores the details necessary to choose an algorithm for the specific operator and to derive a physical plan. In the case of the skyline operator, we use a single node with a single child in the logical plan. This node contains information such as the skyline dimensions SkylineDimension as well as other vital information like whether the skyline should be DISTINCT or not. The child node of the skyline operator node provides the input data for the skyline computation.

Each SkylineDimension extends the default Spark Expression such that it stores both the reference to the database dimension and the type (i.e., MIN/MAX/DIFF). The database dimension is usually a column but can also be a more complex Expression (e.g., an aggregate) in Spark. It is stored as the child of the SkylineDimension which allows us to make use of the generic

functionality responsible for resolving Expressions in the analyzer.

5.3 Extending the Analyzer

The analyzer of Spark already offers a wide range of rules which can be used to resolve the expressions. Since the skyline dimensions used by our skyline operator as well as their respective children are also expressions, they will be resolved automatically by the existing rules in most cases. We have to ensure that all common queries also work when the skyline operator is used. To achieve this, we have to extend the existing rules from the Analyzer to also incorporate the skyline operator. Such rules mainly pertain to the propagation of aggregates across the (unresolved) logical execution plan of Spark.

First, we ensure that we are also able to compute skylines which include dimensions not present in the final projection (i.e., the attributes specified in the SELECT clause). To achieve this, we expand the function ResolveMissingReferences by adding another case for the SkylineOperator. The code is shown in Listing 6. 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 - 3). Then, we resolve the expressions and add the missing attributes (line 4 - 5) which are then used to generate a new set of skyline dimensions (line 6 - 7). If no output was added, we simply replace the skyline dimensions (line 8 - 9). Otherwise, we create a new skyline with the newly generated child (line 10 - 11) and add a projection to eliminate redundant attributes (line 12).

Listing 6: 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)
3        && child.resolved =>
4     val (exprs, newChild) =
5       resolveExprsAndAddMissingAttrs(skylineItems, child)
6     val dimensions =
7       exprs.map(_.asInstanceOf[SkylineDimension])
8     if (child.output == newChild.output) {
9       s.copy(skylineItems = dimensions)
10    } else {
11      val newSkyline = s.copy(dimensions, newChild)
12      Project(child.output, newSkyline)
13    }
```

Next, we need to take care that aggregate attributes are also propagated to the skyline properly. The code to accomplish this

is given in Listing 7. We will only explain it briefly since it was modified from existing Spark code for similar nodes.

To do this, we match the skyline operator which is a parent or ancestor node of an `Aggregate` (line 3). Then we try to resolve the skyline dimensions using the output of the aggregate as a basis (line 5 - 6). This will also introduce missing aggregates in the `Aggregate` node which is necessary, for example, if the output of the query only contains the `sum` but the skyline is based on the `count` (line 7 - 13). Using the output of the resolution, we (re-)construct the skyline operator (line 14 - 17). Additionally, we introduce analogous rules for other similar cases like plans where there is a `Filter` node introduced by a `HAVING` clause between the `Aggregate` and the skyline operator node among others.

Listing 7: Analyzer extension to propagate aggregate attributes to skylines

```
1   case SkylineOperator(
2       distinct, complete, skylineItems,
3       agg: Aggregate
4   ) if agg.resolved =>
5     val maybeResolved = skylineItems.map(_.child)
6       .map(resolveExpressionByPlanOutput(_, agg))
7     resolveOperatorWithAggregate(
8       maybeResolved, agg,
9       (newExprs, newChild) => {
10        val newSkylineItems = skylineItems.zip(newExprs)
11          .map {
12          case (skylineItems, expr) =>
13            skylineItems.copy(child = expr)
14        }
15        SkylineOperator(
16          distinct, complete, newSkylineItems, newChild
17        )
18      }
19    )
```

Note that the correct handling of `Filter` and `Aggregate` nodes is non-trivial in general – not only in combination with the skyline operator. Indeed, when working on the integration of skyline queries into Spark SQL, we noticed that aggregates are in some cases not resolved correctly by the default rules of Spark. This is, for example, the case when `Sort` nodes are resolved in combination with `Filters` and `Aggregates`. Such cases may be introduced by a `HAVING` clause in the query. For a more detailed description of this erroneous behavior of Spark SQL together with a proposal how to fix this bug, see [19].

5.4 Additional Optimizations

Given a resolved logical plan, the *Catalyst Optimizer* aims at improving the logical query plan by applying rule-based optimizations (see Figure 2). The default optimizations of Spark also apply to skyline queries – especially to the ones where the data itself is retrieved by a more complex query. Skyline queries integrated into Apache Spark thus benefit from existing optimizations, which can be further improved by adding the following specialized rules.

If a skyline query contains a single `MIN` or `MAX` dimension, then this is equivalent to selecting the tuples with the lowest or highest values for the skyline dimension respectively. In other words, the Pareto optimum in a single dimension is simply the optimum. There are two possibilities to rewrite the skyline. We can either first sort the values according to the skyline dimension

and then only select the top results or find the lowest or highest value in a scalar subquery (i.e., a subquery that yields a single value) and then select the tuples which have the desired value. Given that, for a relation with n tuples, sorting and selecting exhibits a worst-case runtime of $O(n \cdot \log(n))$ while the scalar subquery and selection can, when optimized, be done in $O(n)$ time, we opt for the latter.

A more sophisticated optimization is taken from [5], where [6] is referenced for the correctness of this transformation. It is applicable if the skyline appears in combination with a `Join` node in the tree. If the output of the join serves as the input to the skyline operator, then we can check whether it is possible to move the skyline into one of the "sides" of the join. This is applicable if the join is *non-reductive* [5] which is defined in [6]. Intuitively, non-reductiveness in our case means that due to the constraints in the database, it can be inferred that for every tuple in the first table joined there must exist at least one "partner" in the other join table if the tuple is part of the skyline. The main benefit of this optimization is that computing the skyline before the join is likely to reduce the input size for both the skyline operator and the (now subsequent) join operation, which typically reduces the total query execution time.

5.5 Algorithm Selection

Given the problems that arise with skyline queries on incomplete datasets as mentioned in Section 3, we have to take extra care which algorithm is used for which dataset. The decision is mainly governed by whether we can assume the input dataset to be complete or not. If the dataset is (potentially) incomplete, we have to select an algorithm that is capable of handling incomplete data since otherwise, the computation may not yield the correct results.

As will be explained in Section 5.7 (and it will also be observed in the empirical evaluation in Section 6), skyline algorithms that are able to handle incomplete datasets are lacking in performance compared to their complete counterparts. Given that Spark can handle multiple different data sources and cannot always detect the `nullability` of a column, it is desirable to provide an override that decides whether a complete algorithm is used regardless of the detected input columns. We do this by introducing the optional `COMPLETE` keyword in the skyline query syntax. This gives the user the possibility to enforce the use of the complete algorithms also on datasets that can technically be incomplete but are known to be complete. The correctness of the algorithm only depends on whether `null` values actually appear in the data.

We implement the algorithm selection using the nodes in the physical execution plan. Which algorithm is selected depends on which physical nodes are chosen during translation of the (resolved and optimized) logical execution plan. In a minimally viable implementation, it is possible to implement the skyline operator in a single physical node. This has, however, the disadvantage that Spark's potential of parallelism would be lost for the skyline computation. We therefore split the computation into two steps, represented by two separate nodes in the physical plan. The pseudocode of the algorithm selection is shown in Listing 8.

The main decision to be made (line 1, 2) is if we may use the complete algorithm. This is the case when the `SKYLINE` clause contains the `COMPLETE` keyword or when the skyline dimensions are recognized as `not nullable` by the system. Note that the local skylines use the same basic nodes both in the complete and incomplete cases, while the nodes for the global skyline differ.

Listing 8: Selection of appropriate skyline nodes in physical execution plan

input : resolved logical plan with skyline operator node sky as root
output: optimized (resolved) logical plan

```
1  skylineNullable ← ∃d ∈ D_SKY : isnullable(d)
2  if COMPLETE is set OR ¬skylineNullable then
3  │   local_skyline ← local_node()
4  │   global_skyline ← complete_global_node()
5  else
6  │   local_skyline ← local_node()
7  │   global_skyline ← incomplete_global_node()
8  return global_skyline
```

The first node in the physical plan corresponds to the *local skyline* computation, which can be done in a distributed manner (line 3 and 6). This means that there may exist multiple instances of the node which can exist distributed across the cluster on which Spark runs. For the actual distribution of the data, we can use multiple different schemes provided by Spark or leave it as-is from the child in the physical plan. When left at standard distribution (`UnspecifiedDistribution`), Spark will usually try to distribute the data equally given the number of available executors. For example, if there are 10 executors available for 10.000.000 tuples in the original dataset, then each executor will receive roughly 1 million tuples each. For the algorithms that can handle incomplete datasets, we have to use a specialized distribution scheme, which will be discussed in more detail in Section 5.7.

The output of the local skyline computation is used as input for calculating the global skyline (line 4 and 7). This means that, in the physical execution plan, we create a global skyline node whose only child is the local skyline. In contrast to the local skyline, we may not freely choose the distribution but must instead ensure that all tuples from the local skyline are handled by the same executor for the global skyline computation. This is enforced by choosing the `AllTuples` distribution provided by Spark.

To keep our solution modular, we encapsulate the *dominance check* in a new utility that we introduce in Spark. It takes as input the values and types of the skyline dimensions of two tuples and checks if one tuple dominates the other. For every skyline dimension, we match the data type to avoid costly casting and (in the case of casting to floating types) potential loss of accuracy.

With our approach to algorithm selection in the physical plan and the modular structure of the code, it is easy to incorporate further skyline algorithms if so desired in the future. Choosing a different algorithm mainly entails replacing the local and/or global skyline computation nodes in the physical plan by new ones.

5.6 Skylines in Complete Datasets

For complete datasets, we adapt the Block-Nested-Loop skyline algorithm [5]. The main idea is to keep a window of tuples in which the skyline of all tuples processed up to this point is stored. We iterate through the entire dataset, and, for each tuple, we check which dominance relationships exist with the tuples in the current window. If tuple t is dominated by a tuple in the window, then t is eliminated. Here, it is not necessary to check t against the remaining tuples since it cannot dominate any tuples in the window due to transitivity. If tuple t dominates one or more

tuples in the window, then the dominated tuples are eliminated. In this case, t is inserted into the window since, by transitivity, t cannot be dominated by other tuples in the window. Tuple t is also inserted into the window if it is found incomparable with all tuples in the current window.

We can use the same algorithm for both the local and the global skyline computation. The only necessary difference is the distribution of the data, where we let Spark handle the distribution for the local skyline while we force the `AllTuples` distribution for the global skyline computation. This has the additional advantage that partitioning done in prior processing steps can be kept, which increases the locality of the data for the local skyline computation and may thus help to further improve the overall performance.

The Block-Nested-Loop approach is most efficient if the window fits into main memory. Note that also Spark, in general, performs best if large portions of the data fit into the main memory available. Especially in cloud-based platforms, there is typically sufficient RAM available (in the order of magnitude of terabytes or even petabytes). At any rate, if RAM does not suffice, Spark will swap data to disk like any other program – with the usual expected performance loss.

5.7 Skylines in Incomplete Datasets

When computing the local skyline for an incomplete dataset, we must take care that no potential cyclic dominance relationships are lost such that tuples may appear in the result even though they are dominated by another tuple. To combat this, we use a special form of a *bitmap*-based skyline algorithm [21].

Given a set P of tuples, we can assign each tuple $p \in P$ a bitmap b (an index in binary format) such that each bit in b corresponds to a skyline dimension. If a tuple p has a null value in a skyline dimension, then the corresponding bit in bitmap b is set to 1; otherwise, it is set to 0. Then, subsequently, the data is partitioned according to the bitmap indexes such that all tuples with a specific bitmap b are assigned to the same subset (partition) P_b of P. We can then calculate the local skyline $SKY(P_b)$ for each set of tuples P_b without losing the transitivity property or running into issues with cyclic dominance relationships.

In Spark, this sort of partitioning is done via the integrated distribution of the nodes. We craft an expression for the distribution which uses the predefined `IsNull()` method to achieve this effect.

For the global skyline computation, we cannot use the standard Block-Nested-Loop approach since cyclic dominance relationships may occur and the transitivity of the skyline operator is not guaranteed. We therefore opt for the less efficient approach where we compare all tuples against each other. Even if a tuple t is dominated, we may not immediately delete it since it may be the only tuple that dominates another tuple t'. In such a case, by prematurely deleting t, tuple t' would be erroneously added to the skyline. This is a subtle point which was, for instance, overlooked in the algorithm proposed in [21]. In [19], we illustrate the error resulting from premature deletion of dominated tuples in detail.

The following lemma guarantees that our skyline computation in case of a potentially incomplete dataset yields the correct result:

LEMMA 5.1. *Let a dataset P be partitioned according to the* null *values and let S_{local} denote the resulting union of local skylines.*

Then it holds for every tuple $p \in P$ not part of the global skyline S_{global} that either $p \notin S_{local}$ or there exists $q \in S_{local}$ with $q \prec p$.

PROOF. Let $p \in P$ be a tuple that is not part of the global skyline S_{global}, i.e., there exists some tuple $q \in P$ with $q \prec p$. If both p and q belong to the same partition during local skyline computation, then this dominance relation will be detected in this step and p will be deleted. Hence, in this case, $p \notin S_{local}$.

Now suppose that p and q belong to different partitions. If $q \in S_{local}$, then q is the desired tuple in S_{local} with $q \prec p$.

It remains to consider the case that $q \notin S_{local}$. This means that q gets deleted during the local skyline computation. In other words, there exists a tuple r in the same partition as q with $r \prec q$ such that $r \in S_{local}$. The latter property is guaranteed by the fact that, inside each partition, all tuples have nulls at the same positions and there can be no cyclic dominance relationships.

By the assumption $q \prec p$ and the definition of skylines in incomplete datasets, q is at least as good as p in all non-missing skyline dimensions and strictly better in at least one. Since $r \prec q$ and both tuples have the same set of missing dimensions due to being in the same partition, it follows that also r is at least as good as p in all non-missing skyline dimensions and strictly better in at least one. That is, r is the desired tuple in S_{local} with $r \prec p$. □

Selecting an algorithm which can handle incomplete datasets yields the correct result also for a complete dataset (while the converse is, of course, not true). There is, however, a major drawback to always using an incomplete algorithm. Since the partitioning in the incomplete algorithm is done according to which values of a tuple are null, only limited partitioning is possible. The worst case occurs when the algorithm for incomplete datasets is applied to a complete dataset. In this case, since there are no null values, there is only a single partition which all tuples belong to. This means that any potential of parallelism gets lost. Hence, in the case of a complete dataset, the algorithm for incomplete data performs even worse than a strictly non-distributed approach which immediately proceeds to the global skyline computation by a single executor.

5.8 Extending the User Interfaces

In addition to our extension of the SQL string-interface of Spark SQL, we integrate the skyline queries also into the basic DataFrame API provided in Java and Scala by adding new API functions. Here, the data about the skyline dimensions can either be passed via pairs of skyline dimension and associated dimension type or by using the *columnar* definition of Spark directly. For the latter purpose, we define the functions smin(), smax(), and sdiff(), which each take a single argument that provides the skyline dimension in Spark columnar format as input. The API calls bypass the parsing step and directly create a new skyline operator node in the logical plan.

We also integrate skyline queries into Python and R via PySpark and SparkR, respectively – two of the most popular languages for analytical and statistical applications. Rather than re-implementing skyline queries in these languages, we build an intermediate layer that calls the Scala-implementation of the DataFrame API.

In Python, this method relies on the external, open-source library Py4J (https://www.py4j.org/). Its usage requires the skyline dimensions and skyline types to first be translated into Java objects, which can then be passed on using Py4J. A slight complication arises from the fact that Python is a weakly typed language, which imposes restrictions on the method signatures. Skyline dimensions and their types are therefore passed as separate lists of strings. The first dimension is then matched to the first type of skyline dimension and so forth. Passing the skyline dimensions via Columns (i.e., Expressions) works similarly to the Scala/Java APIs.

In R, the integration is simpler since R allows for tuples of data to be entered more easily. Hence, in addition to the column-based input analogous to the Python interface, the R interface also accepts a list of pairs of skyline dimension plus type.

5.9 Ensuring Correctness

As far as the impact of the skyline handling on the overall behavior of Spark SQL is concerned, we recall from Sections 5.2 and 5.3 that, in the logical plan, the skyline operator gives rise to a single node with a single input (from the child node) and a single output (to the parent node). When translating the logical node to a physical plan, this still holds since, even if there are multiple physical nodes in the plan, there is only a single input and output.

Therefore, the main concern of ensuring correctness of our integration of skyline queries into Spark SQL is the handling of skyline queries itself. There are no side effects whatsoever of the skyline integration on the rest of query processing in Spark SQL. This also applies to potential effects of the skyline integration on the performance of Spark SQL commands not using the skyline feature at all. In this case, the only difference in the query processing flow is an additional clause in the parser. The additional cost is negligible.

We have intensively tested the skyline handling to provide evidence for its correctness. Unit tests are provided as part of our implementation in the GitHub repository. Additionally, for a significant portion of the experiments reported in Section 6, we have verified that our integrated skyline computation yields the same result as the equivalent "plain" SQL query in the style of Listing 4.

6 EMPIRICAL EVALUATION

In this section, we take a closer look at how our integration performs based on a series of benchmarks executed on a cluster. We will first provide information about the experimental setup as well as the input data and the queries which are executed on them. Following that, we will provide an overview of the outcomes of the benchmark. To round off the section, we will give a brief summary of the performance measurements and draw some conclusions.

6.1 Experimental Setup

We have implemented the integration of skyline queries into Apache Spark, and provide our implementation as open-source code at https://github.com/Lukas-Grasmann/skyline-queries-spark. Our implementation uses the same languages as Apache Spark, which include Java, Scala, Python, and R. The bulk of the main functionality was written in Scala, which is also the language in which the core functionalities of Spark are implemented. The experiments can be easily reproduced by using the provided open-source software. The binaries, benchmark data, queries, and additional scripts can be found at [20].

All tests were run on a cluster consisting of 2 namenodes and 18 datanodes. The latter are used to execute the actual program. Every node consists of 2 Xeon E5-2650 v4 CPUs by Intel that provide 24 cores each, which equals 48 cores per node and up to 864 cores across all worker nodes in total. Each node provides up to 256GB of RAM and 4 hard disks with a total capacity of 4TB each.

The resource management of the cluster is entirely based on Cloudera and uses YARN to deploy the applications. It provides the possibility to access data stored in Hive, which we found convenient for storing and maintaining the test data in our experiments. And it is also a common way how Apache Spark is used in practice.

For fine-tuning the parameters, we use the command line arguments provided by Spark to deploy the applications to YARN. In these tests, we tell Spark how many executors are to be spawned. The actual resource assignment is then left to Spark to provide conditions as close to the real-world setting as possible.

6.2 Test Data and Queries

For our tests with a **real-world dataset**, we use the freely available subset of the Inside Airbnb dataset as input, which contains accommodations from Airbnb over a certain timespan. The time span chosen is 30 days and contains approximately 1 million tuples. The tuples were downloaded from the Inside Airbnb website [1] and then subsequently merged while eliminating duplicates and string-based columns. For the complete variant of the dataset, we have eliminated all tuples containing a null value in at least one skyline dimension. Content-wise, this dataset is similar to the hotel example provided in Section 1. The use cases considered here are also similar and encompass finding the "best" listings according to the chosen dimensions. The relevant key (identifying) dimension and the 6 skyline dimensions can be found in Table 1. From this relation, we constructed skyline queries with 1 dimension, 2 dimensions, …, 6 dimensions by selecting the dimensions in the same order as they appear in Table 1, i.e., the one-dimensional skyline query only uses the first skyline dimension (price) in the table, the two-dimensional query uses the first two dimensions (price, accommodates), etc.

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 1: Skyline dimensions in the Inside Airbnb dataset

We also carried out tests on a **synthetic dataset**, namely the store_sales table from the benchmark DSB [14]. The skyline dimensions used in the benchmarks can be found in Table 2. The table has 2 identifying key dimensions and, again, 6 skyline dimensions. The selection of skyline dimensions to derive 6 skyline queries (with the number of skyline dimensions ranging from 1 to 6) is done exactly as described above for the Airbnb test case.

As far as the test data is concerned, we randomly generated data such that the total size of the table is approximately 15,000,000 tuples (note that data generation in DSB works by indicating the data volume in bytes not tuples; in our case, we generated 1.5 GB of data). For the tests with varying data size, we simply select the first tuples from the table until the desired size of the dataset is reached. For the complete dataset, we only select the tuples which are not null in all six potential skyline dimensions.

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 discount given
ss_ext_sales_price	MIN	sum of sales price

Table 2: Skyline dimensions in the store_sales dataset

For both datasets, there exists a complete as well as an incomplete variant. The difference between them is that for the complete dataset all tuples that contain null values in at least one skyline dimension have been removed. For the real-world dataset, this means that the incomplete dataset is bigger than the complete variant while for the synthetic dataset, both variants have the same size.

6.3 Tested Algorithms

In total, we run our tests on up to four skyline algorithms:

(1) The distributed algorithm for complete datasets (described in Section 5.6), which splits the skyline computation into local and global skyline computation; strictly speaking, only the local part is distributed (whence, parallelized).

(2) The non-distributed complete algorithm, which completely gives up on parallelism, skips the local skyline computation and immediately proceeds to the global skyline computation from the previous algorithm.

(3) The distributed algorithm for incomplete datasets described in Section 5.7. Recall, however, that here, distribution is based on the occurrence of nulls in skyline dimensions, which severely restricts the potential of parallelism.

(4) As the principal reference algorithm for our skyline algorithms, we run our tests also with the rewriting of skyline queries into plain SQL as described in Listing 4.

In our performance charts, we refer to these 4 algorithms as "distributed complete", "non-distributed complete", "distributed incomplete", and "reference". In all tests with complete datasets, we compare all four algorithms against each other. For incomplete datasets, the complete algorithms are not applicable; we thus only compare the remaining two algorithms in these cases.

6.4 Experimental Results

In our experimental evaluation, we want to find out how the following parameters affect the execution time:

- number of skyline dimensions;
- number of input tuples in the dataset;
- number of executors used by Spark.

For the number of skyline dimensions, we have crafted queries with 1 – 6 dimensions. The number of executors used by Spark is chosen from 1, 2, 3, 5, 10. For the size of the dataset, we distinguish between the real-world (Airbnb) and synthetic dataset (DSB): all tests with the real-world dataset are carried out with all tuples contained in the data, i.e., ca. 1,200,000 tuples if `nulls` are allowed and ca. 820,000 tuples after removing all tuples with a `null` in one of the skyline dimensions. In contrast, the synthetic dataset is randomly generated and we choose subsets of size 10^6, $2 \cdot 10^6$, $5 \cdot 10^6$, and 10^7 both, for the complete and the incomplete dataset.

We have carried out tests with virtually all combinations of the above mentioned value ranges for the three parameters under investigation, resulting in a big collection of test cases: complete vs. incomplete data, 1 – 6 skyline dimensions, "all" tuples (for the Airbnb data) vs. varying between 10^6, $2 \cdot 10^6$, $5 \cdot 10^6$, and 10^7 tuples (for the DSB data), and between 1 and 10 executors. In all these cases, we ran 4 algorithms (in the case of complete data) or 2 algorithms (for incomplete data) as explained in Section 6.3. Below we report on a representative subset of our experiments. The runtime measurements are shown in Figures 3 – 7. Further performance charts are provided in [19].

In our experiments, we have defined a timeout of 3600 seconds. Timeouts are "visualized" in our performance charts by missing data points. Actually, timeouts did occur quite frequently – especially with incomplete data and here, in particular, with the "reference" algorithm. Therefore, in Figures 3 – 7, for the case of incomplete data (always the plot on the right-hand side), we only show the results for datasets smaller than 10^7 tuples. In contrast, for complete data (shown in the plots on the left-hand side) we typically scale up to the max. size of 10^7 tuples.

The plots for the impact of the **number of skyline dimensions** on the execution time can be found in Figure 3 for the real-world dataset and Figure 4 for the synthetic dataset. Of course, if the number of dimension increases, the dominance checks get slightly more costly. But the most important effect on the execution time is via the size of the skyline (in particular, of the intermediate skyline in the window).

Increasing the number of dimensions can have two opposing effects. First, adding a dimension can make tuples incomparable, where previously one tuple dominated the other. In this case, the size of the skyline increases due to the additional skyline dimension. Second, however, it may also happen that two tuples had identical values in the previously considered dimensions and the additional dimension introduces a dominance relation between these tuples. In this case, the size of the skyline decreases.

For the real-world data (Figure 3), the execution time tends to increase with the number of skyline dimensions. This is, in particular, the case for the reference algorithm. In other words, here we see the first possible effect of increasing the number of dimensions. In contrast, for the synthetic dataset, we also see the second possible effect. This effect is best visible for the reference algorithm in the left plot in Figure 4. Apparently, there are many tuples with the maximal value in the first dimension (= `ss_quantity`), which become distinguishable by the second dimension (= `ss_wholesale_cost`). For the right plot in Figure 4, it should be noted that the tests were carried out with a 10 times smaller dataset to avoid timeouts, which makes the test results less robust. The peak in the case of two dimensions is probably due to "unfavorable" partitioning of tuples causing some of the local skylines to get overly big.

At any rate, when comparing the algorithms, we note that the specialized algorithms are faster in almost all cases and scale

significantly better than the "reference" algorithm (in particular, in case of the real-word data). For complete data, the "distributed complete" algorithm performs best. The discrepancy between the best specialized algorithm and the reference algorithm is over 50% in most cases and can get up to ca. 80% (complete synthetic data, 5 dimensions) and even over 95% (complete synthetic data, 1 dimension).

The impact of the **size of the dataset** on the execution time is shown in Figure 5. Recall that we consider the size of the real-world dataset as fixed. Hence, the experiments with varying data size were only done with the synthetic data. Of course, the execution time increases with the size of the dataset, since a larger dataset also increases the number of dominance checks needed. The exact increase depends on the used algorithm; it is particularly dramatic for the "reference" algorithm, which even reaches the timeout threshold as bigger datasets are considered. Again, the "distributed complete" algorithm, when applicable, performs best and, in almost all cases, all specialized skyline algorithms outperform the "reference" algorithm. The discrepancy between the best specialized algorithm and the reference algorithm reaches ca. 60% for incomplete data and over 80% for complete data (for $5 \cdot 10^6$ tuples). For the biggest datasets (10^7 tuples), the reference algorithm even times out.

The impact of the **number of executors** on the performance is shown in Figure 6 for the real-world dataset and in Figure 7 for the synthetic dataset. Ideally, we want the algorithms to be able to use an arbitrary number of executors productively. Here, we use the term "executors" since it is the parameter by which we can instruct Spark how many instances of the code it should run in parallel.

We observe that the benefit of additional executors tapers off after a certain number of executors has been reached. This point is reached sooner, the smaller the dataset is. This behavior, especially in the case of the distributed algorithms, can be explained as follows: As we increase the parallelism of the local skyline computation, the portion of data contained in each partition becomes smaller; this means that more and more dominance relationships get lost and fewer tuples are eliminated from the local skylines. Hence, more work is left to the global skyline computation, which allows for little to no parallelism and, thus, becomes the bottleneck of the entire computation. In other words, there is a sweet spot in terms of the amount of parallelism relative to the size of the input data.

Note that the "reference" algorithm is also able to make (limited) use of parallelism. Indeed, the execution plan generated by Spark from the plain SQL query is still somewhat distributed albeit not as much as the truly distributed and specialized approaches. As in the previous experiments, the "reference" algorithm never outperforms any of the specialized algorithms – not even the non-distributed or quasi-non-distributed (incomplete algorithm on complete dataset) approaches. The discrepancy between the best specialized algorithm and the reference algorithm is constantly above 50% and may even go up to 90% (complete synthetic data, 10 executors).

6.5 Further Measurements and Experiments

Our main concern with the experiments reported in Section 6.4 was to measure the execution time of our integrated skyline algorithms depending on several factors (number of dimensions/tuples/executors). Apart from the execution time, we made further
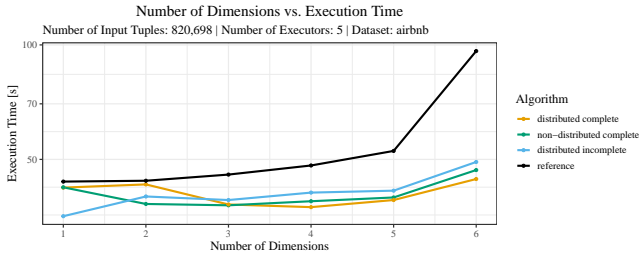
Figure 3: Number of dimensions vs. execution time on the Inside Airbnb dataset
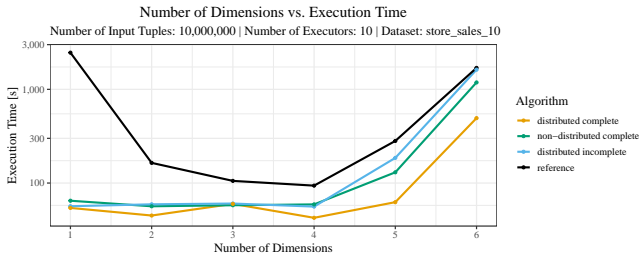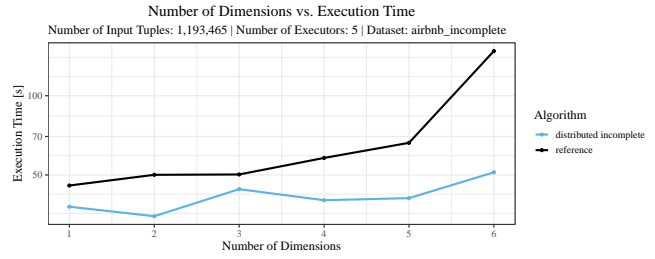

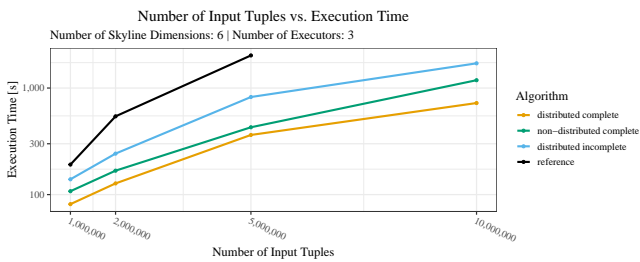**Figure 4: Number of dimensions vs. execution time on the store_sales dataset**
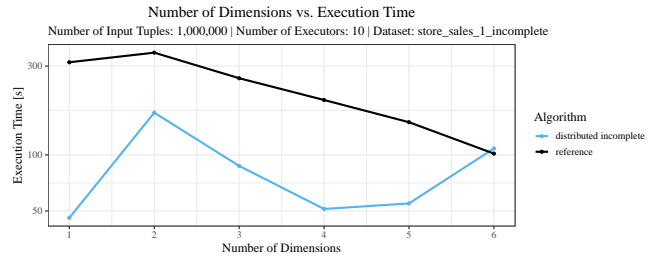

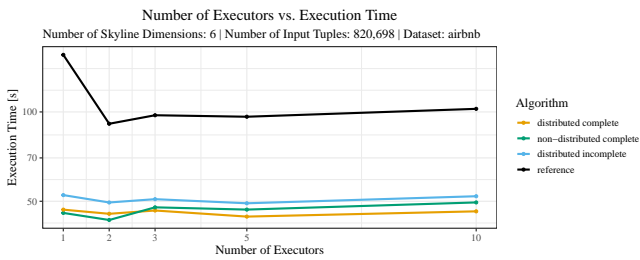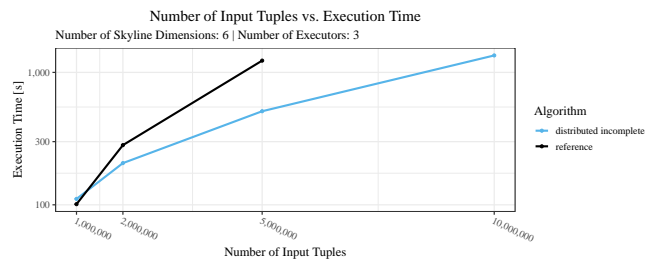**Figure 5: Number of input tuples vs. execution time on the store_sales dataset**

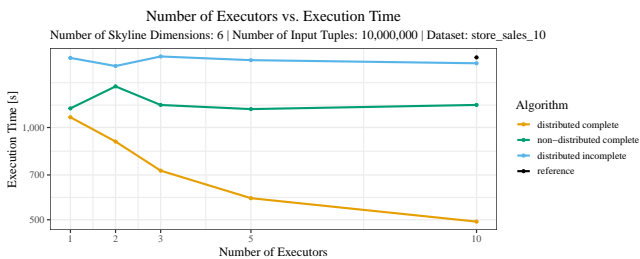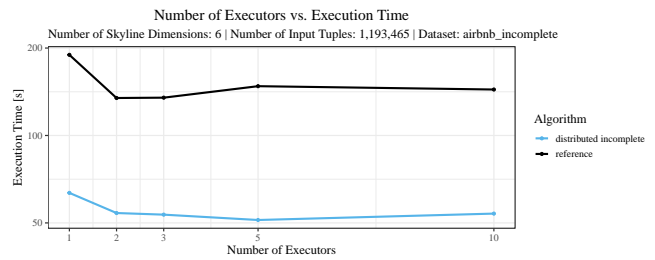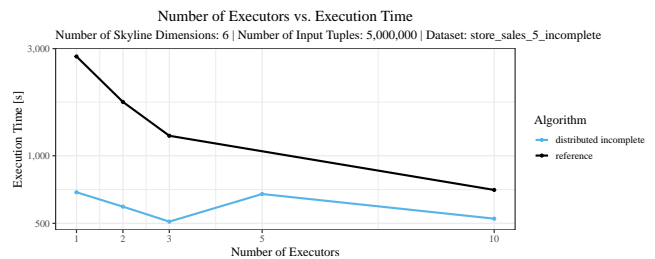**Figure 6: Number of executors vs. execution time on the Inside Airbnb dataset**

Figure 7: Number of executors vs. execution time on the store_sales dataset

measurements in our experiments. In particular, we were interested in the **memory consumption** of the various algorithms in the various settings. In short, the results were unsurprising: the number of dimensions does not influence the memory consumption while the number of executors and, even more so, the number of input tuples does. In the case of the executors this is mainly due to the fact that each executor loads its entire execution environment, including the Java execution framework, into main memory. In the case of the number of tuples, of course, more memory is needed if more tuples are loaded. Importantly, we did not observe any significant difference in memory consumption between the four algorithms studied here. For the sake of completeness, we provide plots with the memory consumption in various scenarios in [19].

The test setup described in Section 6.2 was intentionally minimalistic in the sense that we applied skyline queries to base tables of the database rather than to the result of (possibly complex) SQL queries. This allowed us to focus on the behavior of our new skyline algorithms vs. the reference queries without having to worry about side-effects of the overall query evaluation. But, of course, it is also interesting to test the behavior of skyline queries on top of complex queries. To this end, we have carried out experiments on the MusicBrainz database [16] with more complex queries including joins and aggregates. Due to lack of space, the detailed results are given in [19]. In a nutshell, the results are quite similar to the ones reported in Section 6.4:

- The execution time of the reference solution is almost always above the time of the specialized algorithms. The only cases where the reference solution performs best are the easiest ones with execution times below 50 seconds. For the harder cases it is always slower and – in contrast to the specialized algorithms – causes several timeouts.
- The memory consumption is comparable for all 4 algorithms. There are, however, some peaks in case of the reference algorithms that do not occur with the specialized algorithm. In general, the behavior of the reference algorithm seems to be less stable than the other algorithms. For instance, with 4, 5, or 6 dimensions, the execution time jumps from below 50 seconds for 3 executors to over 1000 seconds for 5 executors and then slightly decreases again for 10 executors.
- It is also notable, that skyline queries rewritten to plain SQL are much less readable and less intuitive in the context of the entire query.

6.6 Summary of Results

From our experimental evaluation, we conclude that, in almost all cases, our specialized algorithms outperform the "reference" algorithm and they also provide better scalability. Parallelization (if applicable) has proved profitable – but only up to a certain point, that depends on the size of the data. Moreover, it has become clear that using the incomplete distributed algorithm on a complete dataset is not advisable and may perform worse than the non-distributed algorithm. As such, the introduction of additional syntax to select an appropriate algorithm through the keyword COMPLETE may help to significantly boost the performance.

7 CONCLUSION AND FUTURE WORK

In this work, we have presented the extension of Spark SQL by the skyline operator. It provides users with a simple, easy to use syntax for formulating skyline queries. To the best of our knowledge, this is the first distributed data processing system with native support of skyline queries. By our extension of the various components of Spark SQL query execution and, in particular, of the Catalyst optimizer, we have obtained an implementation that clearly outperforms the formulation of skyline queries in plain SQL. Nevertheless, there is still ample space for future enhancements:

So far, we have implemented only the Block-Nested-Loop skyline algorithm and variants thereof. It would be interesting to implement additional algorithms based on other paradigms like ordering [3, 4, 10, 11, 17, 18, 34] or index structures [29, 30, 41] and to evaluate their strengths and weaknesses in the Apache Spark context. Also, for the partitioning scheme, further options such as angle-based partitioning [26, 43] are worth trying. Further specialized algorithms, that require a deeper modification of Spark (such as Z-order partitioning [42], which requires the computation of a Z-address for each tuple [31]) are more long-term projects.

For our skyline algorithm that can cope with (potentially) incomplete datasets, we have chosen a straightforward extension of the Block-Nested-Loop algorithm. As mentioned in Section 5.7, this may severely limit the potential of parallelism. Moreover, in the worst case, the algorithm may thus have to compare each tuple against any other tuple. Clearly, a more sophisticated algorithm for potentially incomplete datasets would be highly desirable.

We have included *rule-based optimizations* in our integration of skyline queries. The support of *cost-based optimization* by Spark SQL is only somewhat rudimentary as of now. Fully integrating skyline queries into a future cost-based optimizer will be an important, highly non-trivial research and development project. However, as soon as further skyline algorithms are implemented, a light-weight form of cost-based optimization should be implemented that selects the best-suited skyline algorithm for a particular query.

Finally, from the user perspective, the integration into different Spark modules such as structured streaming would be desirable.

To conclude, the goal of this work was a full integration of skyline queries into Spark SQL. The favorable experimental results with simple skyline algorithms (Block-Nested-Loops and variants thereof) demonstrate that an integrated solution is clearly superior to a rewriting of the query on SQL level. To leverage the full potential of the integrated solution, the implementation of further (more sophisticated) skyline algorithms is the most important task for future work. It should be noted that, in our implementation, we have paid particular attention to a modular structure of our new software so that the implementation of further algorithms is possible without having to worry about Spark SQL as a whole. Moreover, our code is provided under Apache Spark's own open-source license, and we explicitly invite other groups to join in this effort of enabling convenient and efficient skyline queries in Spark SQL.

REFERENCES

[1] Inside Airbnb. 2022. Inside Airbnb. Retrieved January 19, 2022 from http://insideairbnb.com/get-the-data.html

[2] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. 2010. Selecting skyline services for QoS-based web service composition. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti (Eds.). ACM, 11–20. https://doi.org/10.1145/1772690.1772693

[3] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2006. SaLSa: computing the skyline without scanning the whole sky. In *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*, Philip S. Yu, Vassilis J. Tsotras, Edward A. Fox, and Bing Liu (Eds.). ACM, 405–414. https://doi.org/10.1145/1183614.1183674

[4] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2008. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.* 33, 4 (2008), 31:1–31:49. https://doi.org/10.1145/1412331.1412343

[5] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 421–430. https://doi.org/10.1109/ICDE.2001.914855

[6] Michael J. Carey and Donald Kossmann. 1997. On Saying "Enough Already!" in SQL. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, Joan Peckham (Ed.). ACM Press, 219–230. https://doi.org/10.1145/253260.253302

[7] Bee-Chung Chen, Raghu Ramakrishnan, and Kristen LeFevre. 2007. Privacy Skyline: Privacy with Multidimensional Adversarial Knowledge. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 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 (Eds.). ACM, 770–781. http://www.vldb.org/conf/2007/papers/research/p770-chen.pdf

[8] Lei Chen and Xiang Lian. 2008. Dynamic skyline queries in metric spaces. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings (ACM International Conference Proceeding Series)*, Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu (Eds.), Vol. 261. ACM, 333–343. https://doi.org/10.1145/1353343.1353386

[9] Jan Chomicki, Paolo Ciaccia, and Niccolò Meneghetti. 2013. Skyline queries, front and back. *SIGMOD Rec.* 42, 3 (2013), 6–18. https://doi.org/10.1145/2536669.2536671

[10] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. 2003. Skyline with Presorting. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman (Eds.). IEEE Computer Society, 717–719. https://doi.org/10.1109/ICDE.2003.1260846

[11] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. 2005. Skyline with Presorting: Theory and Optimizations. In *Intelligent Information Processing and Web Mining, Proceedings of the International IIS: IIPWM'05 Conference held in Gdansk, Poland, June 13-16, 2005 (Advances in Soft Computing)*, Mieczyslaw A. Klopotek, Slawomir T. Wierzchon, and Krzysztof Trojanowski (Eds.), Vol. 31. Springer, 595–604. https://doi.org/10.1007/3-540-32392-9_72

[12] Bin Cui, Hua Lu, Quanqing Xu, Lijiang Chen, Yafei Dai, and Yongluan Zhou. 2008. Parallel Distributed Processing of Constrained Skyline Queries by Filtering. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 546–555. https://doi.org/10.1109/ICDE.2008.4497463

[13] Evangelos Dellis and Bernhard Seeger. 2007. Efficient Computation of Reverse Skyline Queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 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 (Eds.). ACM, 291–302. http://www.vldb.org/conf/2007/papers/research/p291-dellis.pdf

[14] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[15] Hannes Eder. 2009. *On Extending PostgreSQL with the Skyline Operator*. Master's thesis. Fakultät für Informatik der Technischen Universität Wien.

[16] MetaBrainz Foundation. 2022. MusicBrainz. Retrieved August 30, 2022 from https://musicbrainz.org/

[17] Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2005. Maximal Vector Computation in Large Data Sets. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 229–240. http://www.vldb.org/archives/website/2005/program/paper/tue/p229-godfrey.pdf

[18] Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2007. Algorithms and analyses for maximal vector computation. *VLDB J.* 16, 1 (2007), 5–28. https://doi.org/10.1007/s00778-006-0029-7

[19] Lukas Grasmann, Reinhard Pichler, and Alexander Selzer. 2022. Integration of Skyline Queries into Spark SQL. *CoRR* arXiv:2210.03718 (2022). https://doi.org/10.48550/arXiv.2210.03718

[20] Lukas Grasmann, Reinhard Pichler, and Alexander Selzer. 2022. Integration of Skyline Queries into Spark SQL - Experiment Artifacts. Zenodo. https://doi.org/10.5281/zenodo.7157840

[21] Yonis Gulzar, Ali Amer Alwan, and Sherzod Turaev. 2019. Optimizing Skyline Query Processing in Incomplete Data. *IEEE Access* 7 (2019), 178121–178138. https://doi.org/10.1109/ACCESS.2019.2958202

[22] Katja Hose and Akrivi Vlachou. 2012. A survey of skyline processing in highly distributed environments. *VLDB J.* 21, 3 (2012), 359–384. https://doi.org/10.1007/s00778-011-0246-6

[23] Xuegang Huang and Christian S. Jensen. 2004. In-Route Skyline Querying for Location-Based Services. In *Web and Wireless Geographical Information Systems, 4th International Workshop, W2GIS 2004, Goyang, Korea, November 2004, Revised Selected Papers (Lecture Notes in Computer Science)*, Yong Jin Kwon, Alain Bouju, and Christophe Claramunt (Eds.), Vol. 3428. Springer, 120–135. https://doi.org/10.1007/11427865_10

[24] Christos Kalyvas and Theodoros Tzouramanis. 2017. A Survey of Skyline Query Processing. *CoRR* abs/1704.01788 (2017), 127. arXiv:1704.01788 http://arxiv.org/abs/1704.01788

[25] Aki Kanellis. 2020. SkySpark. Retrieved November 10, 2020 from https://github.com/AkiKanellis/skyspark

[26] Junsu Kim and Myoung Ho Kim. 2018. An efficient parallel processing method for skyline queries in MapReduce. *J. Supercomput.* 74, 2 (2018), 886–935. https://doi.org/10.1007/s11227-017-2171-y

[27] Donald Kossmann, Frank Ramsak, and Steffen Rost. 2002. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 275–286. https://doi.org/10.1016/B978-155860869-6/50032-9

[28] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route skyline queries: A multi-preference path planning approach. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE Computer Society, 261–272. https://doi.org/10.1109/ICDE.2010.5447845

[29] Jongwuk Lee and Seung-won Hwang. 2010. BSkyTree: scalable skyline computation using a balanced pivot selection. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings (ACM International Conference Proceeding Series)*, Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan (Eds.), Vol. 426. ACM, 195–206. https://doi.org/10.1145/1739041.1739067

[30] Jongwuk Lee and Seung-won Hwang. 2010. BSkyTree: scalable skyline computation using a balanced pivot selection. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings (ACM International Conference Proceeding Series)*, Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan (Eds.), Vol. 426. ACM, 195–206. https://doi.org/10.1145/1739041.1739067

[31] Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, and Yuan Tian. 2010. Z-SKY: an efficient skyline query processing framework based on Z-order. *VLDB J.* 19, 3 (2010), 333–362. https://doi.org/10.1007/s00778-009-0166-x

[32] Ken C. K. Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. 2007. Approaching the Skyline in Z Order. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 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 (Eds.). ACM, 279–290. http://www.vldb.org/conf/2007/papers/research/p279-lee.pdf

[33] Xin Lin, Jianliang Xu, and Haibo Hu. 2011. Authentication of location-based skyline queries. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 1583–1588. https://doi.org/10.1145/2063576.2063805

[34] Rui Liu and Dominique Li. 2020. Efficient Skyline Computation in High-Dimensionality Domains. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 459–462. https://doi.org/10.5441/002/edbt.2020.57

[35] Michael D. Morse, Jignesh M. Patel, and H. V. Jagadish. 2007. Efficient Skyline Computation over Low-Cardinality Domains. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 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 (Eds.). ACM, 267–278. http://www.vldb.org/conf/2007/papers/research/p267-morse.pdf

[36] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2003. An Optimal and Progressive Algorithm for Skyline Queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 467–478. https://doi.org/10.1145/872757.872814

[37] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *ACM Trans. Database Syst.* 30, 1 (2005), 41–82. https://doi.org/10.1145/1061318.1061320

[38] Ioanna Papanikolaou. 2018. *Distributed Algorithms for Skyline Computation using Apache Spark.* Master's thesis. INTERNATIONAL HELLENIC UNIVERSITY. https://repository.ihu.edu.gr/xmlui/handle/11544/29524 Accessed on: 23.01.2022.

[39] Yoonjae Park, Jun-Ki Min, and Kyuseok Shim. 2013. Parallel Computation of Skyline and Reverse Skyline Queries Using MapReduce. *Proc. VLDB Endow.* 6, 14 (2013), 2002–2013. https://doi.org/10.14778/2556549.2556580

[40] Cheng Sheng and Yufei Tao. 2012. Worst-Case I/O-Efficient Skyline Algorithms. *ACM Trans. Database Syst.* 37, 4 (2012), 26:1–26:22. https://doi.org/10.1145/2389241.2389245

[41] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. 2001. Efficient Progressive Skyline Computation. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 301–310. http://www.vldb.org/conf/2001/P301.pdf

[42] Mingjie Tang, Yongyang Yu, Walid G. Aref, Qutaibah M. Malluhi, and Mourad Ouzzani. 2019. 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*. IEEE, 2113–2114. https://doi.org/10.1109/ICDE.2019.00251

[43] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. 2008. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 227–238. https://doi.org/10.1145/1376616.1376642

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, Erich M. Nahum and Dongyan Xu (Eds.). USENIX Association, 10. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[45] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[46] Shiming Zhang, Nikos Mamoulis, and David W. Cheung. 2009. Scalable skyline computation using object-based space partitioning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 483–494. https://doi.org/10.1145/1559845.1559897