Efficient Execution Plans for Distributed Skyline Query Processing

João B. Rocha-Junior*, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{joao,vlachou,cdoulk,noervaag}@idi.ntnu.no

ABSTRACT

In this paper, we study the generation of efficient execution plans for skyline query processing in large-scale distributed environments. In such a setting, each server stores autonomously a fraction of the data, thus all servers need to process the skyline query. An execution plan defines the order in which the individual skyline queries are processed on different servers, and influences the performance of query processing. Querying servers consecutively reduces the amount of transferred data and the number of queried servers, since skyline points obtained by one server prune points in the subsequent servers, but also increases the latency of the system. To address this trade-off, we introduce a novel framework, called *SkyPlan*, for processing distributed skyline queries that generates execution plans aiming at optimizing the performance of query processing. Thus, we quantify the gain of querying consecutively different servers. Then, execution plans are generated that maximize the overall gain, while also taking into account additional objectives, such as bounding the maximum number of hops required for the query or balancing the load on different servers fairly. Finally, we present an algorithm for distributed processing based on the generated plan that continuously refines the execution plan during in-network processing. Our framework consistently outperforms the state-of-the-art algorithm.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

Skyline query, distributed systems, execution plan

*On leave from the Universidade Estadual de Feira de Santana.

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

1. INTRODUCTION

The importance of skyline queries [4] has been recognized for applications that involve decision-making. Skyline queries do not require an explicit preference function, which may be difficult for the user to define when the relative importance of the different criteria is vague. Consider for example a real-estate database containing information about houses, represented as tuples. Each tuple stores different characteristics of a house, for instance its price and its distance to a point of interest, such as the nearest metro station. A person that is interested in a house that is cheap and close to a metro station poses a skyline query to retrieve the best offers. The houses in the skyline set are those that are not worse than (*not dominated by*) any other house in both price and distance.

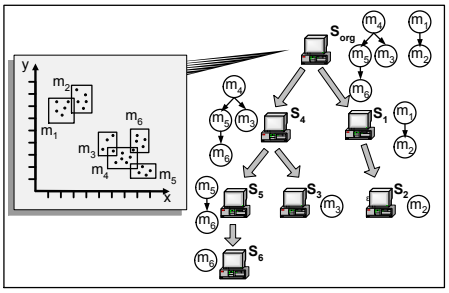


Figure 1: Distributed skyline query processing.

Skyline queries have been studied both in centralized and distributed environments [3–5,7,10,13,14,16,18,20–24]. However, planning the execution of skyline queries on different (potentially overlapping) data fractions, in order to obtain the skyline set of the entire dataset efficiently, has not received adequate attention in the related work. This problem is particularly important and challenging in distributed environments, where each server stores only a fraction of the available data. Moreover, with the advent of large-scale data centers and cloud computing infrastructures, data is increasingly stored and processed in a distributed way. Thus, it is necessary to efficiently support complex query types, such as skyline queries, in distributed environments.

Figure 1 depicts an example of a dataset distributed over multiple servers, S_1 to S_6. The query originator S_{org} can directly communicate with any other server. Each server S_i stores autonomously a fraction of the data that in this example correspond to the data points enclosed in the rectangle m_i. In order to retrieve the skyline set of the distributed

dataset, all servers compute the skyline set of the locally stored data points. Then, S_{org} gathers these points and computes the skyline set of the entire dataset. Before processing the distributed query, S_{org} first derives an *execution plan* that defines the order in which the query is processed on the different servers.

The execution plan influences the overall performance of the query. For example, depending on the execution plan, some servers do not have to be contacted at all. In Figure 1, S_6 can be removed from the execution plan after processing the skyline query at S_4, since there exists a point stored locally at S_4 that dominates all points stored at S_6. Furthermore, the number of data points transferred back to S_{org} can be drastically reduced, by discarding data points stored at S_i that are dominated by skyline points of servers that precede S_i in the execution plan. In addition, the efficiency of local query processing at S_i can be improved by using the skyline points of previous servers to immediately discard points stored locally at S_i. On the other hand, the execution of local skyline queries on consecutive servers is a blocking operation and may lead to increased response time. Motivated by this discussion, in this paper, we study the problem of creating efficient execution plans that improve the performance of the distributed skyline query.

Our main observation is that a *dependency* must exist between consecutive servers, in order to obtain any performance gain by reducing the number of contacted servers or transferred data. A dependency between two servers means that points of the first server must dominate at least some points of the following server. Therefore, our approach optimizes the performance of distributed skyline computation, by exploiting the dependencies – when they exist – between skyline queries on different servers. The dependencies between skyline queries are complex, and their representation forms a directed graph with cycles. For example, if the servers S_3 and S_4 are considered, both local skyline queries depend on each other. Thus, it is not feasible to incorporate all dependencies in the execution plan, and the major challenge is to derive an execution plan that includes the most promising dependencies. Our proposed framework, called *SkyPlan*, quantifies the efficiency of an execution plan based on the potential gain obtained by querying different servers consecutively (henceforth this gain is mentioned as *pruning power*), and generates *cost-aware* execution plans. In more details, we make the following contributions:

- We propose SkyPlan, a framework that generates cost-aware execution plans with maximum pruning power and drastically reduces the response time of distributed skyline query processing.
- We extend our framework to produce multi-objective execution plans, when additional objectives – other than maximizing the pruning power – need to be fulfilled simultaneously.
- We propose an efficient algorithm that exploits the generated execution plan for processing skyline queries in a distributed environment. We further enhance the performance of query execution by continuously refining both the execution plan and the filter points, as the query is being processed in a distributed manner.
- Finally, we perform an extensive experimental evaluation that demonstrates that SkyPlan consistently outperforms the state-of-the-art algorithm [7] by 1-3 orders of magnitude.

The remaining of this paper is organized as follows: Section 2 overviews the related work. Then, we present the necessary preliminaries in Section 3 and the overview of SkyPlan in Section 4. In Section 5 the dependency graph is defined, while in Section 6 the quality of a cost-aware execution plan is introduced and the plan generation algorithm is presented. In Section 7, we describe multi-objective execution plans that satisfy additional constraints. Section 8 presents the distributed skyline computation guided by the execution plan. Finally, the experimental evaluation is presented in Section 9 and we conclude in Section 10.

2. RELATED WORK

Skyline computation [4] has recently attracted considerable attention and has been studied in a variety of distributed systems, including web information systems [3], parallel systems [17], peer-to-peer systems [5, 10, 13, 18, 19, 21–23], mobile ad-hoc networks [14, 20], as well as more generic distributed systems [7, 16, 24]. In the following, we provide an overview of existing approaches for distributed skyline computation. One of the first approaches, by Balke *et al.* [3], focuses on skyline query processing over multiple sources, with each source storing only a subset of attributes (vertical data distribution). Afterwards, most of the existing approaches focus on highly distributed environments, such as peer-to-peer networks, assuming that all data sources store common attributes (horizontal data distribution). Such approaches can be classified in two categories.

In the first category, the proposed methods assume space partitioning among servers, thus each server is responsible for a disjoint partition of the data space. Several approaches belong to this category, namely DSL [23], SSP [21], SkyFrame [22] and iSky [5]. DSL was proposed by Wu *et al.* [23] and it is the first paper that addresses constrained skyline query processing over disjoint data partitions by using a structured peer-to-peer overlay, namely CAN. Wang *et al.* [21] propose the SSP algorithm based on the use of a tree-based peer-to-peer overlay (BATON) for assigning data to servers. Later, the authors present SkyFrame [22] as an extension of their work. Chen *et al.* [5] propose the iSky algorithm, which employs an alternative transformation, namely iMinMax, in order to use the BATON overlay.

In the second category, data partitioning is assumed and each server autonomously stores its own data. Hose *et al.* [13] use distributed data summaries for efficient processing of approximate skyline queries and provide guarantees for the completeness of the result. Skyline computation over a super-peer architecture has been studied in [18,19]. SKYPEER [18] transforms the multi-dimensional data into one-dimensional values and utilizes a thresholding scheme in order to reduce the transferred data. SKYPEER+ [19] extends SKYPEER, by focusing on efficient routing of skyline queries over the super-peer network. Fotiadou *et al.* [10] propose BITPEER for supporting efficiently continuous subspace skylines in a distributed setting by using distributed bitmap indexes. Huang *et al.* [14] study skyline query processing over mobile ad-hoc networks. In [20], bandwidth-constrained skyline queries in mobile environments are studied.

In addition, a few methods have been proposed that assume data partitioning among servers without the restriction

of an existing overlay network, i.e., the query originator can directly communicate with all servers. Cui *et al.* [7] proposed the PaDSkyline algorithm, where the data stored at each server are summarized by MBRs. Initially, the MBRs of all servers are collected and partitioned to incomparable groups. For each group of MBRs, an intra-group query execution order is defined and the authors propose two strategies. The first is the linear execution of queries on servers, where the MBRs are sorted based on their Euclidean distance to the origin of the data space. The second strategy is to build a tree that defines the execution order, but the generated tree is strongly dependent on the order that the MBRs were collected and processed. Assuming the same architecture, in [16], an approach called AGiDS is proposed that uses a grid-based data summary of the data stored locally at each server for defining the execution order. In [24], a feedback-based distributed skyline (FDS) algorithm is proposed, which aims to minimize the bandwidth consumption at the expense of several round-trips.

Our framework, similarly to [7, 16, 24], makes no assumption on the existence of a specific overlay network. Differently than [7], we focus on the execution plan which influences the overall performance of the distributed query processing. In contrast, the execution order proposed in [7] does not take into account the gain of querying two servers consecutively and thus, does not optimize the performance of query processing. In addition, we propose a distributed query execution mechanism that dynamically eliminates parts of the plan and constantly refines the filter points. Furthermore, SkyPlan uses MBRs for data summarization in contrast to AGiDS [16] that uses a grid-based data summary. The main shortcoming of AGiDS is that it assumes that the partitions of the grid-based data summary are common and known a-priori to all servers, which is not feasible for large and highly dynamic distributed systems. Moreover, AGiDS does not produce tree-based execution plans, in contrast to SkyPlan. Finally, SkyPlan focuses on minimizing the response time, which is different than the goal of FDS [24], since FDS requires several round-trips to process the query, thus it incurs high response time.

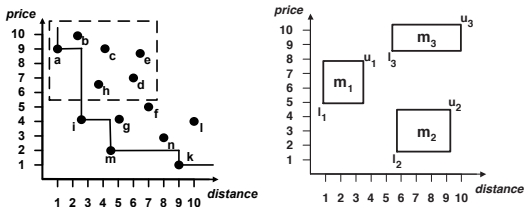
3. PRELIMINARIES

Given a dataset D on a data space defined by a set of d dimensions $\{d_1, ..., d_d\}$, a point $p \in D$ is represented as p=$\{p_1, ..., p_d\}$ where p_i is the value on dimension d_i. Without loss of generality, we assume that $\forall d_i : p_i \geq 0$, and that smaller values are preferable.

DEFINITION 1. *A point $p \in D$ dominates another point $q \in D$, denoted as $p \prec q$, if (1) on every dimension d_i, $p_i \leq q_i$; and (2) on at least one dimension d_j, $p_j < q_j$. The skyline is a set of points SKY which are not dominated by any other point in D.*

Consider the example in Figure 2(a), where each point represents a hotel and the y-dimension represents the price of a room, while the x-dimension captures the distance of the hotel to the beach. A hotel dominates another hotel because it is cheaper and closer to the beach. Thus, the skyline points (a, i, m and k) are the best possible tradeoffs between price and distance from the beach.

The notion of skyline queries can be extended to *constrained skyline queries*. Given a set of constraints, a constrained skyline query returns the skyline points by considering only the data points enclosed in the data space defined by the constraints. Each constraint is expressed as a range along a dimension and the conjunction of all constraints forms a hyper-rectangle in the d-dimensional data space. For example, the dashed rectangle in Figure 2(a) defines a constrained skyline query and the constrained skyline points are a and h. A hyper-rectangle $m_i(l_i, u_i)$ is represented by two points, its lower left corner l_i and its upper right corner u_i. The dominance relationships between two hyper-rectangles m_i and m_j are described as follows:



(a) Skyline example. (b) Domination example.

Figure 2: Skyline and domination examples.

DEFINITION 2. *Given two hyper-rectangles m_i and m_j, the dominance relationships are: (1) m_i dominates m_j, if $u_i \prec l_j$; (2) m_i partially dominates m_j, if $l_i \prec u_j$, but $u_i \not\prec l_j$; (3) m_i and m_j are incomparable, if $l_i \not\prec u_j$ and $l_j \not\prec u_i$.*

For example in Figure 2(b), m_1 and m_2 are incomparable, which means that no point enclosed in m_1 can dominate any point in m_2 and vice versa. Moreover, m_1 dominates m_3 (since u_1 dominates l_3), while m_2 partially dominates m_3 (since l_2 dominates u_3 but u_2 does not dominate l_3). We also mention that a point p dominates a hyper-rectangle $m_i(l_i, u_i)$ if and only if $p \prec l_i$, i.e., point p dominates the lower left corner of m_i.

As stated in [7], the volume of the dominance area quantifies the pruning power of a data point. The *dominance area* of a point p is defined as the area that is enclosed by the hyper-rectangle that has as lower left corner the point p and as upper right corner the maximum corner of the universe. Furthermore, given a hyper-rectangle m_i, we introduce the notion of *enclosed dominance area* of a point p, as the area within the hyper-rectangle m_i that is dominated by p. Similarly, given two hyper-rectangles m_i and m_j, the *enclosed dominance area* V_{ij} of m_i on m_j is the volume of m_j that is dominated by the lower left corner l_i of m_i. Based on the above notions, we can define the pruning power PP_{ij} of m_i on m_j.

DEFINITION 3. *Given two hyper-rectangles m_i and m_j, the pruning power PP_{ij} of m_i on m_j is defined as $PP_{ij} = \frac{V_{ij}}{V_j}$, where V_j denotes the volume of m_j and V_{ij} denotes the enclosed dominance area of m_i on m_j.*

4. SKYPLAN FRAMEWORK

In our system model, a set of $|S|$ servers S_i participate in the distributed skyline computation and each server S_i can directly connect to any other server S_j. Each server S_i stores locally a set of points D_i. The entire dataset D is

the union of all sets of points D_i stored locally at any server S_i ($D = \bigcup D_i$). A skyline query can be initiated by any server, henceforth also called query originator S_{org}. In such a distributed system, a skyline query is processed by sending the query to all servers S_i, which in turn process the query locally over their data D_i. Then, each server S_i reports its *local skyline set* SKY_i to S_{org} for subsequent merging (discarding of dominated local skyline points), in order to obtain the *global skyline set* SKY. The correctness of the result is ensured by the property that the skyline points over a horizontally partitioned dataset are a subset of the union of the skyline points of all partitions ($SKY \subseteq \bigcup SKY_i$).

Optimizing the performance of distributed skyline computation requires defining an appropriate *execution plan*. The execution plan defines the order in which the individual skyline queries are processed on different servers. By querying the servers consecutively, some servers may not have to be contacted at all, if all points of a server are dominated by a point stored locally at another server. Furthermore, the amount of transferred data can be drastically reduced. The local skyline points (or a fraction of them called *filter points* [7, 14] $F_i \subseteq SKY_i$) of a server S_i can be used for discarding dominated points at another server S_j. In such a filter-based approach, the servers are accessed based on an execution order and the filter points of server S_i are selected and transferred to the next servers S_j to discard local skyline points produced by S_j. Filter points not only significantly reduce the amount of transferred data that need to be merged, but also the processing time at the subsequent servers by discarding points with few dominance tests only. However, the filter points may fail to prune any point of a server depending on the data distribution. When no gain can be obtained from querying the servers consecutively, the parallelism should be preserved, in order to minimize the latency and therefore also the response time[1]. To address this tradeoff, this paper introduces a novel framework, called SkyPlan, for processing distributed skyline queries that generates execution plans aiming at optimizing the performance of query processing.

Creation of efficient execution plans in distributed systems requires that the query originator S_{org} has at least an abstract knowledge of the data stored on each server. Therefore, each server S_i reports a set of minimum bounding rectangles[2] (MBRs) to S_{org} as a summarization of its data. Each MBR reported by a server S_i defines a constrained skyline query on S_i, where the constraints are set by the boundaries of the MBR. Thus, S_{org} generates an execution plan that defines the order of the constrained skyline queries on different servers.

S_{org} can immediately discard dominated MBRs, since the local skyline points that belong to those MBRs are definitely dominated by points of other servers. The remaining MBRs are partitioned into *incomparable groups*, such that any two MBRs that belong to different groups are incomparable. Each incomparable group is processed in parallel, since no gain can be achieved by filtering. Unfortunately, the MBRs usually form only a few incomparable groups and thus, the number of MBRs in a group is high. This is because the probability of partial dominance between any two MBRs is high and it increases with dimensionality. Therefore, the major challenge is establishing the execution order for the MBRs that belong to the same group. SkyPlan maps the dominance relationships between MBRs to a weighted directed graph, called skyline dependencies graph (SD-graph). The weights on the graph edges are defined by the pruning power, which is used to quantify the potential gain through filtering. Then, the SD-graph is transformed into an execution plan that maximizes the total pruning power, while preserving the parallelism, when no significant gain can be obtained from processing the queries on different servers consecutively. Furthermore, SkyPlan supports multi-objective executions plans, in case that additional objectives – other than maximizing the pruning power – need to be fulfilled simultaneously.

To summarize, SkyPlan consists of three phases: S_{org} collects the MBRs of all servers S_i and builds a weighted directed graph (Section 5); the graph is transformed into an execution plan that defines the order of query execution on servers (Section 6 and Section 7); and the plan is executed in a distributed manner that enables refinement of both the execution plan and the filter points during query processing (Section 8).

5. SKYLINE DEPENDENCIES GRAPH

In this section, we first provide the necessary definition of the graph that captures the dependencies between the skyline queries. Thereafter, we present the construction algorithm for the dependencies graph.

5.1 Definition

Given two constrained skyline queries defined by the MBRs m_i and m_j, we say that there exists a dependency between them, if some points of m_i may be dominated by points of m_j or vice-versa. Thus, the dominance relationships between any pair of MBRs also define the dependency between the corresponding constrained skyline queries. SkyPlan maps the complex dependencies among a set of non-dominated MBRs to a graph, mentioned as *skyline dependencies graph* or *SD-graph* for short.

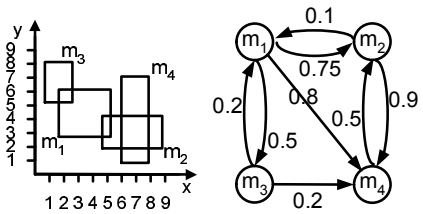
DEFINITION 4. ***SD-graph*** G. *Given a set of MBRs, the SD-graph is the weighted directed graph* $G = (N, E, w)$, *where:*

- N *is the set of nodes and each node* $n_i \in N$ *corresponds to a non-dominated MBR* m_i.
- E *is the set of edges.* E *is a set of ordered pairs* $e_{ij} = (i, j)$, *where* m_i, m_j *are MBRs, and* m_i *partially dominates* m_j.
- w *is a weight function defined as:* $w_{ij} = \frac{|m_j|}{|D|} PP_{ij}$, *where* $|m_j|$ *denotes the number of points enclosed in* m_j, $|D|$ *the cardinality of the dataset* D *and* PP_{ij} *denotes the pruning power of* m_i *on* m_j.

Each non-dominated MBR m_i is represented as a node (vertex) n_i of the graph. Dominated MBRs are immediately discarded, as they enclose data points that are dominated and cannot belong to the skyline set. For each MBR m_j that is partially dominated by m_i, a weighted directed edge e_{ij} from node n_i to n_j is added. Consider the set of MBRs

[1]The response time is the time that passes between the query is posed until the results can be reported back to the user.

[2]An MBR is a hyper-rectangle and any point of S_i is enclosed by at least one MBR.



(a) Set of MBRs. (b) SD-graph.

Figure 3: Construction of SD-graph.

depicted in Figure 3(a). In this example, m_1 partially dominates m_2, because the lower left corner of m_1 dominates the upper right corner of m_2. Thus, a directed edge from m_1 to m_2 is added to the graph (Figure 3(b)). Note that we use node n_i and MBR m_i interchangeably in our examples. The weight w_{ij} of the edge e_{ij} is defined as the pruning power (PP_{ij}) of m_i on m_j normalized by the number of points $|m_j|$ enclosed in m_j divided by the cardinality of the entire dataset $|D|$. The normalization is due to the fact that if m_i has the same pruning power on two different MBRs, then the dependency between m_i and the MBR that has more enclosed data points is more important for the overall performance of the distributed query.

5.2 Construction of the SD-Graph

The SD-graph is constructed by comparing each pair of MBRs. If an MBR is dominated, its node is removed from the graph. If an MBR is partially dominated, then an edge is added between the corresponding nodes.

Algorithm 1 presents our algorithm for the construction of the SD-graph. Given a set of MBRs, our algorithm constructs the SD-graph incrementally, by adding to the graph one MBR in each iteration. Initially, the sets of nodes and edges of the SD-graph are empty. Then, the algorithm takes as input the current form of the SD-graph G and a new MBR m_n, which should be accommodated in the existing graph G. Thus, a new node is added to the graph G that represents the MBR m_n (line 3). Then, each node of G (corresponding to MBR m_i) is compared against m_n for dominance, in order to discard dominated MBRs (lines 5,9). If m_n is dominated by or dominates m_i, then the dominated MBR is discarded. Once an MBR is dominated, all edges connected to the discarded MBR are also eliminated.

Otherwise, m_n and m_i are tested for partial dominance, in order to determine potential dependencies and add the corresponding edges. If m_i partially dominates m_n (line 14), an edge from m_i to m_n is created (line 15). In the same spirit, if m_n partially dominates m_i (line 17), a directed edge from m_n to m_i is added (line 18). Note that when an edge is added to the graph, the weight of the edge is also computed. For sake of simplicity, this is not explicitly shown in the algorithm. At the end of this process, the constructed SD-graph accurately captures all dependencies between any pair of MBRs.

6. COST-AWARE EXECUTION PLANS

In this section, we first provide the definitions of execution plan and its quality in terms of pruning power. Then, we present an algorithm which maps the SD-graph into the execution plan that has the maximum pruning power.

Algorithm 1 *ConstructGraph(G,m_n)*

```
1: INPUT: SD-graph G = (N, E), new MBR m_n.
2: OUTPUT: The updated SD-graph G = (N, E)
3: N ← N ∪ m_n
4: for (∀m_i ∈ N) do
5:    if (m_i dominates m_n) then
6:       N ← N − m_n
7:       E ← E − {e_⋆n} − {e_n⋆}
8:       exit
9:    else if (m_n dominates m_i) then
10:      N ← N − m_i
11:      E ← E − {e_⋆i} − {e_i⋆}
12:      continue
13:   end if
14:   if (m_i partially dominates m_n) then
15:      E ← E ∪ e_in
16:   end if
17:   if (m_n partially dominates m_i) then
18:      E ← E ∪ e_ni
19:   end if
20: end for
```

6.1 Definitions

During distributed skyline query computation an execution plan defines the order of execution of the constrained skyline queries on the individual servers.

DEFINITION 5. ***Execution plan.*** *Given a distributed skyline query, an execution plan $P(N, E)$ is a set of directed rooted weighted trees (forest), where the nodes N represent the individual constrained skyline queries processed on different servers and the weights of edges E represent the pruning power associated with pairs of skyline queries.*

A first observation is that the execution plan is either a tree or a set of trees. The edges have a natural orientation away from the roots. Each tree is processed in parallel and the children of a node are processed after the parent node. Thus, the fan-out of a node defines the degree of parallelism, while the child-parent relation reduces the communication cost of the skyline computation via filtering. The weight that is assigned to an edge $e_{ij} \in E$ indicates the strength of the dependency between the corresponding queries and quantifies the gain of processing the skyline query on $n_j \in N$ after $n_i \in N$. As discussed in Section 5, the pruning power of a edge is an appropriate measure to quantify the potential gain of consecutive queries. Thus, the *quality of an execution plan* is defined as follows.

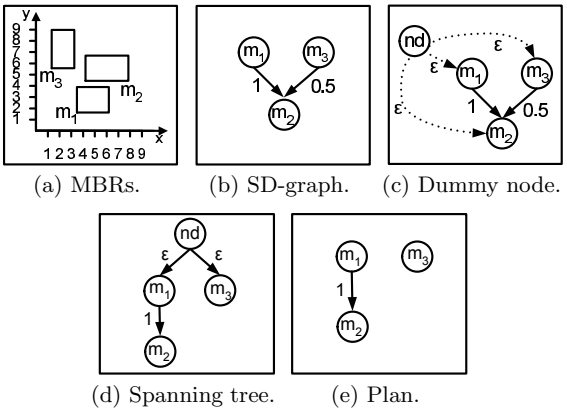
DEFINITION 6. ***Quality of execution plan.*** *The quality $Q(P)$ of an execution plan $P(N, E)$ is defined as the sum of weights on the edges of P:*

$$Q(P) = \sum_{e_{ij} \in E} w_{ij}$$

Thus, an execution plan P_i is better than another plan P_j in terms of quality if: $Q(P_i) > Q(P_j)$.

A straightforward execution plan, also called *linear* in [7], is to execute queries in a sequential manner, by defining an appropriate ordering of queries[3]. Even though the linear approach reduces the transferred data through filter points, it does not exploit parallelism at all, and fails to process in parallel even queries that are independent. Furthermore, the dependencies among the constrained skyline queries form a

[3]In [7] the linear execution plan uses the Euclidean distance from the origin for sorting.



(a) MBRs. (b) SD-graph. (c) Dummy node.

(d) Spanning tree. (e) Plan.

Figure 4: Example of dummy node.

complex graph (SD-graph) that contains cycles and multiple incoming edges to some nodes. Therefore, the mapping of the SD-graph to any execution plan leads to substantial loss in terms of overall pruning power, as several edges cannot be taken into account. Thus, even for a linear execution plan, the efficiency in terms of pruning power is not guaranteed. In the following, we define an execution plan that is generated from the SD-graph and induces sequential processing of queries with the most significant (in terms of pruning power) dependencies, and parallelizes query execution in other cases.

Notice that the SD-graph can in general be disconnected and therefore may consist of more than one connected components (mentioned as subgraphs), such that no edge connects any two nodes that belong to different subgraphs. In this case, the connected subgraphs of the SD-graph correspond to the incomparable groups of MBRs (as discussed in Section 4). Therefore, an individual plan is created (for each subgraph), which is executed in parallel and their results are reported independently. Henceforth, we assume that the SD-graph is connected and present our algorithm for generating query execution plans. Obviously, if the SD-graph is disconnected, the algorithm is applied on each connected subgraph of the SD-graph.

6.2 Maximizing the Pruning Power

Based on Definition 6, an execution plan is better than another execution plan, if its quality in terms of pruning power is higher. Consequently, the ideal execution plan is the execution plan with the maximum pruning power.

DEFINITION 7. ***Maximum pruning execution plan.*** *Let* Π *denote the set of all possible execution plans. The maximum pruning execution plan* $\hat{P}$ *is the one that maximizes the quality of any possible execution plan* $P \in \Pi$*:*

$$\hat{P} = arg\max_{P \in \Pi} Q(P)$$

The problem of generating the maximum pruning execution plan from the SD-graph is equivalent to the *maximum spanning tree problem*, which is formulated as follows.

PROBLEM 1. ***The Directed Maximum Spanning Tree Problem.*** *Consider a directed graph,* $G(N, E)$*, where* N *and* E *are the sets of nodes and edges respectively. Associated with each edge* $e_{ij} \in E$ *is a weight* w_{ij}*. Given a node* n_r*, the problem is to find a rooted directed spanning tree* $P(N, E')$ *with root* n_r*, where* E' *is a subset of* E*, such that the sum of* w_{ij} *for all* $e_{ij} \in E'$ *is maximized. The rooted directed spanning tree is defined as an acyclic graph that connects all nodes using* $|N| - 1$ *edges, which means that each node (excluding the root) has one and only one incoming edge.*

A prerequisite for the existence of a directed rooted spanning tree of the SD-graph is that all nodes must be reachable from the root node, which means that there must exist a directed path from the root to any other node n_i. This may not hold for any node in the SD-graph, even though the SD-graph is connected. Consider the small example with three MBRs depicted in Figure 4(a). The SD-graph has two edges (Figure 4(b)), i.e., $m_1 \rightarrow m_2$ and $m_3 \rightarrow m_2$. For this graph, there does not exist a directed tree that connects all nodes, no matter which node is selected as a root. Therefore, it is not possible to create an execution plan that consists of one directed tree from this graph and the only feasible solution is a set of directed rooted trees.

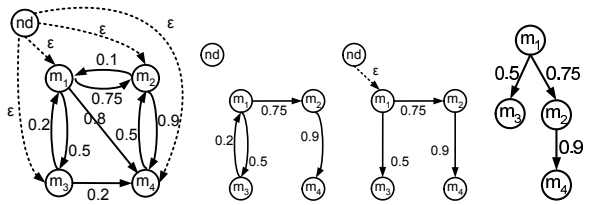
In order to ensure that there always exists a solution for the directed maximum spanning tree problem, we use the following technique. A dummy node *nd* is added to the graph, which is directly connected with all other nodes. All (dummy) edges that start from the dummy node have the same weight, equal to a very small value ε (Figure 4(c)). By using the dummy node as the root for the maximum spanning tree problem, there always exists a directed rooted tree that is a solution of the problem, since all nodes become reachable from the root. Furthermore, a dummy edge is selected only in the case where there exists no directed spanning tree with fewer dummy edges. After the generation of the spanning tree (Figure 4(d)), the dummy node is discarded from the execution plan. After the removal of the dummy node, the execution plan consists of two trees m_3 and $m_1 \rightarrow m_2$ (Figure 4(e)). Notice that the potential pruning of m_2 by data points in m_3 cannot be established, since the execution plan cannot maintain all dependencies, due to the cycles and multiple incoming edges that exist in the SD-graph. The definition of the maximum spanning tree problem guarantees that the produced execution plan has the highest pruning power as there exists no other tree-based plan with higher value of pruning power. The usage of the dummy node ensures that there always exists a spanning tree and also solves the problem of finding an appropriate node for the root of the spanning tree.

Edmonds algorithm. To derive the maximum directed spanning tree of a graph G, we employ the algorithm of Edmonds [6,8]. In the sequel, we briefly sketch Algorithm 2. The algorithm takes as input a node v_{root} that is the root of the spanning tree. In our case, the root of the spanning tree is always the dummy node. First, for each node, except for the root, the incoming edge with the maximum weight is selected. Therefore, $|N| - 1$ edges are selected in set S, as many as the edges of the spanning tree (lines 3-6). If there does not exist a cycle, the algorithm terminates by returning the spanning tree. Otherwise, the main idea of the algorithm is to add those edges that lead to minimum loss in terms of weights, compared to the weights of the edges that form the cycle. Thus, for each cycle, an edge e_{kj} of the cycle pointing to a node v_j is removed and replaced by an unselected edge ($e_{ij} \in E$) pointing to v_j (line 15). To this end, all edges pointing to a node that belongs to the cycle are examined and inserted in set M (lines 9-13). The edge

Algorithm 2 *GenerateExecutionPlan($G(V,E)$,v_{root})*

1: **INPUT:** SD-graph $G(V,E)$, v_{root} the root.
2: **OUTPUT:** Spanning tree $T(V,S)$
3: **for** $(\forall v_i \in V - \{v_{root}\})$ **do**
4: $e : max(w_{ji})$, $\forall j$ /* incoming maximum weighted edge */
5: $S \leftarrow S \cup e$
6: **end for**
7: **while** ($\exists cycle$ in $T(V,S)$) **do**
8: $w_{max} \leftarrow max$ weight of any edge in $cycle$
9: $M \leftarrow \emptyset$
10: **for** $(\forall e_{ij} \in E$ such that $v_j \in cycle$ and $v_i \notin cycle)$ **do**
11: $w_{ij} \leftarrow w_{ij} + (w_{max} - w_{kj}), e_{kj} \in cycle$
12: $M \leftarrow M \cup e_{ij}$
13: **end for**
14: $e \leftarrow e_{ij} \in M$ with minimum loss in weight
15: update S by replacing $e_{kj} \in cycle$ with e
16: replace $cycle$ in G with a pseudo-node
17: **end while**
18: **return** $T(V,S)$

in M that minimizes the loss in weights is selected (line 14), which is the edge with maximum w_{ij} (as computed in line 11). Furthermore, in each iteration the cycle is replaced by a pseudo-node (line 16) and the weights pointing to the cycle are modified (line 11). This ensures that parts of the graph G that have been mapped to a subtree by removing a cycle, are not examined further. The algorithm terminates when all cycles are eliminated. This algorithm is a polynomial-time algorithm for finding a maximum directed spanning tree. A more efficient implementation of the algorithm with complexity $O(|E| + |N|log|N|)$ is presented in [11].



(a) Dummy node. (b) Step 1. (c) Step 2. (d) Plan.

Figure 5: Construction of the execution plan.

EXAMPLE 1. *Figure 5 depicts the process of generating the execution plan that corresponds to the SD-graph of Figure 3(b). First, the dummy node is added to the SD-graph (Figure 5(a)). Then, in the first iteration of Algorithm 2 the edges depicted in Figure 5(b) are selected. Since there exists a cycle, the edge pointing from m_3 to m_1 is replaced with the edge from nd to m_1 (Figure 5(c)). Afterwards, Algorithm 2 terminates, since there exists no other cycle. Finally, the dummy node is removed leading to the execution plan depicted in Figure 5(d). Based on the definition of the maximum spanning tree for the given SD-graph, there exists no other execution plan with higher sum of the edges' weights than the depicted execution plan. The node m_1, which has a high dependency to m_2 and m_3, is the root node of the execution plan. On the other hand, the MBRs m_2 and m_3 are processed in parallel without any loss of pruning power since they are incomparable. Nevertheless, in order to break the cycles that exist in the SD-graph, points of m_1 cannot be pruned by points of m_2 nor m_3 based on the execution plan, since m_1 has a higher pruning power on m_2 and m_3.*

7. MULTI-OBJECTIVE EXECUTION PLANS

Depending on the characteristics or the current load of a distributed system, the generation of execution plans that satisfy additional objectives, apart from maximizing the total pruning power, is desirable. Such an objective is to restrict the number of hops that the query has to be forwarded, so that the latency of the system is bounded and the overall execution time is improved. Another objective is to share the processing load fairly to the servers, by producing balanced execution plans, so that bottlenecks and single points of failure are avoided. In this section, we extend SkyPlan to a generic framework that supports a variety of execution plans in order to adapt to the requirements of different distributed systems.

7.1 k-Hop Execution Plan

A major concern in many distributed systems is bounding the number of hops that are required for query processing. This is particularly important for many applications that require low latency during query processing. The number of required hops is determined by the height of the execution plan. To bound the number of hops, we define the *k-hop execution plan* and show that the SkyPlan framework can be adapted to support such plans efficiently.

DEFINITION 8. ***k-hop execution plan.*** *Given a constraint of k hops, the k-hop execution plan $\hat{P}_k$ is the execution plan with height at most k ($height(\hat{P}_k) \leq k$) that has the maximum quality $Q(\hat{P}_k)$ among all execution plans with height at most k.*

The problem of producing a k-hop execution plan is equivalent to the *hop-constrained maximum spanning tree problem* that produces the spanning tree with height at most k that has the maximum sum of edges' weights. This problem is NP-Hard [12], therefore we employ a heuristic algorithm to produce the k-hop execution plan efficiently. Specifically, we adapt a solution proposed by Abdalla *et al.* [1] for the diameter-constrained minimum spanning tree problem, since the hop-constrained maximum spanning tree problem is a simplification of the bounded-diameter minimum spanning tree problem [15]. Notice that the dummy node added to the SD-graph ensures that there always exists at least one spanning tree with maximum height k ($k \geq 1$).

The algorithm that builds the k-hop execution plan works as follows. First, the maximum pruning execution plan is created by applying Algorithm 2 on the SD-graph. If the length of the longest path is at most k, then the generated execution plan is also the k-hop execution plan and our algorithm terminates. Otherwise, the generated execution plan is modified by reducing the height of the tree iteratively until its maximum path length is at most k. During each iteration, the tree is traversed and the longest path of the tree is detected. Then, the edge with minimum pruning power (excluding dummy edges) in the path is replaced with another edge of SD-graph that reduces the path in at least one hop. Notice that in our case any edge of the plan can be replaced by a dummy edge that reduces the length of the path, because the dummy node is connected to all nodes. If more than one edge that reduce the length of the path exist, then the edge with the highest pruning power is selected. In the next iteration, the current longest path is selected and its length is reduced, until the length of the longest path is at most k.

7.2 Capacity Constrained Execution Plan

Another desirable feature of distributed query processing is to balance the load fairly to all participating servers. Load balancing is important in order to avoid bottlenecks in the system when the query workload increases. To this end, we define the *capacity constrained execution plan* that aims to balance the load, and we describe the process of generating such balanced plans by SkyPlan.

DEFINITION 9. ***Capacity constrained execution plan.*** *Given a capacity b, an execution plan satisfies the capacity constraint, if every subtree s of the root node has at most b nodes ($size(s) \leq b$). The capacity constrained execution plan $\hat{P}_b$ is the execution plan that has the maximum quality $Q(\hat{P}_b)$ among all execution plans that satisfy the capacity constraint.*

The problem of producing a capacity constrained execution plan is equivalent to the problem of computing a *capacitated maximum spanning tree* (CMST) [2]. In the general case of CMST, both edges and nodes have weights and the capacity of a subtree is the sum of the weights of all nodes in the subtree. The CMST is the problem of finding the spanning tree with maximum quality that satisfies the capacity constraint for all subtrees. One interesting instance of the CMST problem is when all weights of the nodes are set to one (except of the weight of the root node that is set to zero). This problem is known as equal weight CMST and it has also been shown to be NP-hard [2]. The equal weight CMST problem can be employed for finding the capacity constrained execution plan. We employ an approximate algorithm [9] to obtain the execution plan efficiently. The existence of the dummy node again guarantees that the SD-graph can always produce such a plan.

The algorithm starts with generating the maximum spanning tree using Algorithm 2 and then checks if the produced execution plan satisfies the capacity constraint. If the constraint is satisfied, no modification is needed and the algorithm terminates. If not, the algorithm separates the nodes n_i of the generated execution plan in two sets A and B. The set A contains all nodes that are in subtrees that satisfy the capacity constraint, while the set B contains the remaining nodes. The algorithm then selects the edge e_{ij} with smallest weight such that $n_j \in B$. This edge is replaced with another edge e_{lj} of the SD-graph such that (a) $n_l \in A$, and (b) the replacement causes the reduction of the overall capacity of the plan. The algorithm terminates when all subtrees have at most b nodes.

8. DISTRIBUTED QUERY PROCESSING

Any server in the system can pose a skyline query and the querying server is referred to as query originator S_{org}. The query execution starts with S_{org} requesting in parallel the MBRs of all servers S_i. Each S_i reports a set of MBRs to S_{org}, and while the MBRs from the servers S_i are received, S_{org} builds the SD-graph $G(N, E)$ using the incremental construction algorithm (Algorithm 1). The incremental property of the algorithm ensures that S_{org} does not need to wait until it receives all MBRs, but starts as soon as it receives MBRs from any server S_i. After receiving all MBRs and building the SD-graph, an appropriate execution plan is established as described in Sections 6 and 7.

Algorithm 3 *QueryProcessing(S_i,F,P)*

```
1: INPUT: Filter points F = {f_1, ..., f_k},
          Execution plan P.
2: OUTPUT: Local skyline
3: m ← P.getRootMBR()
4: sky ←computeSkyline(m)
5: P' ←refinePlan(P, sky)
6: F' ←refineFilters(F, sky)
7: S' ← P'.getNextServers()
8: for (∀S_j ∈ S') do
9:    sky_j ←QueryProcessing(S_j, F',P'_j)
10: end for
11: sky ←mergeSkyline(sky, sky_j)
12: return sky
```

The distributed skyline query is processed based on the execution plan. The query originator S_{org} sends a skyline query to the root of every directed tree in the execution plan. The query sent to a server S_i consists of the execution plan and a set of k filter points $F = \{f_1, \ldots, f_k\}$. Initially, the set of filter points F is empty.

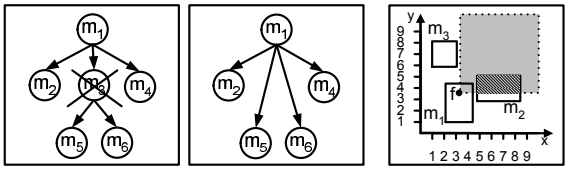
8.1 Query Execution and Plan Refinement

Query execution on a server S_i is initiated when the execution of the plan reaches an MBR m_j of S_i. This initiates a constrained skyline query on m_j. Server S_i takes as input a set of filter points F and the execution plan[4]. S_i produces as output the local skyline points, computes a new set of filter points and refines the execution plan, as will be explained shortly. Then, S_i forwards the query to the servers that are next based on the refined execution plan.

Algorithm 3 provides the pseudocode for query processing on server S_i that received a skyline request. Initially, S_i processes locally the constrained skyline query (line 4) defined by the MBR m that is the root of execution plan P (line 3). The locally computed skyline set with constraint m is denoted as sky. Then, S_i refines the execution plan (line 5), by removing MBRs that are dominated by one of the local skyline points sky. Notice that S_i also removes the current MBR from the execution plan. In addition, S_i is able to refine the filter points F' (line 6), by taking into account the local skyline points and the filter points in the set F. Finally, S_i determines the next set of servers S' that should process the query based on the refined plan P' (line 7), and sends a skyline query to each of them (line 9), passing as arguments the refined filter points F' and the refined plan P'. Each server S_j receives a different execution plan P'_j that corresponds to the part of the execution plan that has as a root the MBR of the server S_j. Eventually, S_i gathers the local result sets of the servers S_j (that had received the query through S_i) and merges them by discarding dominated points (line 11). Local processing on server S_i terminates by returning the local merged skyline points to the previous server based on the execution plan.

The refinement of the execution plan P at server S_i produces a new execution plan P' that does not contain the nodes (representing MBRs) that are dominated by the local skyline points of S_i. In order to maintain the connectivity of the execution plan, when a node is removed, its children nodes substitute it in the plan. For an illustrative example see Figure 6(a), which depicts an execution plan P (left)

[4]Henceforth, we will refer to the subtree of the execution plan that S_i receives as execution plan at S_i.



(a) Plan refinement example. (b) Filter points.

Figure 6: Plan refinement and filter points.

and a refined plan P' (middle). Assume that server S_i determines that m_3 is dominated by local results and should therefore be discarded. In the refined execution plan P', the children MBRs of m_3, namely m_5 and m_6, substitute m_3.

Essentially, as the execution plan is forwarded to servers during query processing, a distributed query execution mechanism is established, which is not coordinated rigidly by one server. Instead, it is a loose approach in which each server is assigned not only the task of local query processing, but also the refinement of the execution plan and the continuation of query processing.

8.2 Filter Point Selection and Refinement

SkyPlan discovers an appropriate execution order to execute the constrained skyline queries on servers S_i by using the pruning power. The filter points facilitate pruning of local data points, hence the selection of the k filter points $F = \{f_1, \ldots, f_k\}$ influences the overall performance. The filter points are selected from the points that belong to local result set and the previous filter points. We refer to these points as candidate filter points.

In [7], the volume of the dominance area is used to estimate the pruning power of a potential filter point. The main shortcoming of this approach is that the volume of the dominance area does not necessarily relate to the area within an MBR that is dominated by a filter point. For example, in Figure 6(b), even though filter point f has a high volume of dominance area, the actual area dominated in m_2 is much smaller. Furthermore, for different MBRs, different filter points may be appropriate. Again in Figure 6(b), point f is not appropriate for m_3 since it cannot dominate any point enclosed in m_3, although it prunes points in m_2.

In order to alleviate such deficiencies, we propose two different methods for filter point selection that rely on the enclosed dominance area of filter point f. In Figure 6(b), the enclosed dominance area of filter point f on m_2 is depicted as the dashed area. Obviously, the enclosed dominance area quantifies the pruning power of a candidate filter point in a more accurate way by taking into account the MBRs of the execution plan.

Our filter strategies discover the filter points that maximize the pruning power based on the queries that will be processed in the next step. Therefore, given an execution plan, the MBRs corresponding to the children of the current node are taken into consideration. The first strategy selects as refined filter points the set of k candidate points that maximize the total enclosed dominance area for all children MBRs. Therefore, for each candidate filter point f_i, the volume of enclosed dominance area is computed for each child MBR m_j and the total volume is defined as the sum of the volumes. Thus, the first strategy selects the same set of filter points for all children MBRs of the current node in the execution plan. The second strategy refines the filter points even further, by defining a different set F_i of filter points for each child MBR. In this case, for each MBR m_i the k filter points with the highest volume of the enclosed dominance area in m_i are selected.

9. EXPERIMENTAL EVALUATION

In this section, we provide an experimental evaluation on the performance of SkyPlan. SkyPlan was implemented in Java, while the network aspects were simulated using DesmoJ[5], an event-based simulator framework. All experiments were conducted on a PC equipped with a 3GHz Dual Core AMD processor and 2GB RAM. The parameters and values used in our experiments are outlined in Table 1 (the values in bold are the default values).

Parameter	**Values**
Dimensions	3, **4**, 5, 6
Number of servers	**1K**, 2K, 3K, 4K
Cardinality of each server	**1K**, 2K, 3K, 4K
Data distributions	RL, **CL**, UN, CO, AC
Network speed (Mbit/s)	0.05, **0.2**, 0.8, 3.2, 12.8

Table 1: Experimental parameters and values.

We employed both synthetic and real datasets. For the synthetic datasets we examined different distributions, namely uniform (UN), clustered (CL), correlated (CO), and anticorrelated (AC). For the clustered dataset (CL), each server picks cluster centroids randomly and the points follow a Gaussian distribution on each axis with variance 0.025, and a mean equal to the corresponding coordinate of the centroid. The correlated (CO) and anticorrelated (AC) datasets were generated as described in [4]. For our experiments on synthetic data, we report the average results over 20 different instances of the data set. We generate the different instances by keeping the parameters fixed and changing the seeds of the random number generator. We adopt this approach in order to factor out the effects of randomization. The real dataset (RL) contains information about real estate all over the United States (crawled from *http://www.zillow.com/*). It is a 5-dimensional dataset containing more than 2M entries with the following attributes: number of bathrooms, number of bedrooms, living area, price and lot area. In all cases, the dataset is horizontally partitioned evenly among the servers. In our experiments, we set the maximum number of filter points k equal to 5, but k does not exceed the 10% of the local skyline points.

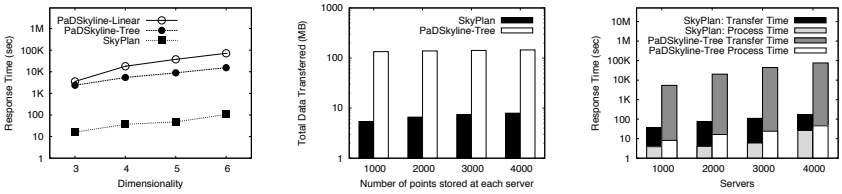
Our main metrics are: (i) the response time, which is the total time until the final result is produced at S_{org} (including the time for generating the execution plan), and (ii) the amount of transferred data. In addition, we measure the Data Reduction Rate (DRR), which has been used in both [7] and [14]. The DRR at a server S_i is measured using the following equation:

$$DRR_i = \frac{|SKY_i| - |SKY_i{}^R| - |F_i|}{|SKY_i|}$$

where $|SKY_i|$ is the cardinality of the skyline of S_i without filtering, $|SKY_i{}^R|$ is the cardinality of the skyline of S_i after filtering, and $|F_i|$ is the cardinality of the filter point set received by S_i. The DRR quantifies the gain in transferred data that is obtained when filter points are used.

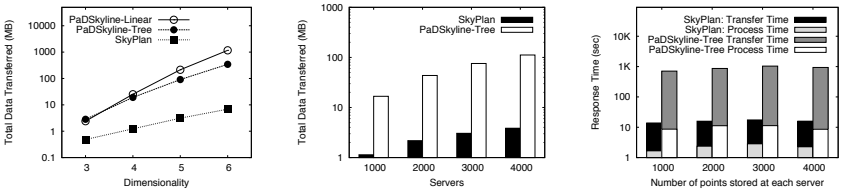
In Section 9.1, we compare *SkyPlan* against the state-of-the-art algorithm (PaDSkyline [7]), which has two variants:

[5]http://desmoj.sourceforge.net/home.html



(a) Time vs. dimensionality. (b) Data vs. cardinality. (c) Time vs. servers.

Figure 7: Comparative performance of SkyPlan against PaDSkyline for uniform (UN) data distribution.



(a) Data vs. dimensionality. (b) Data vs. servers. (c) Time vs. cardinality.

Figure 8: Comparative performance of SkyPlan against PaDSkyline for clustered (CL) data distribution.

Linear and *Tree*. PaDSkyline-Tree constructs an execution plan in which the order of accessing servers depends on the order that the MBRs are received by S_{org}. We implemented PaDSkyline in the same framework as SkyPlan with recursive data transfer according to the execution plan. In addition, we evaluate the different filter point strategies in Section 9.2, and compare the performance of the multi-objective execution plans to the maximum pruning execution plan in Section 9.3. We emphasize that the maximum pruning execution plan is used as the default execution plan of SkyPlan in our experimental evaluation.

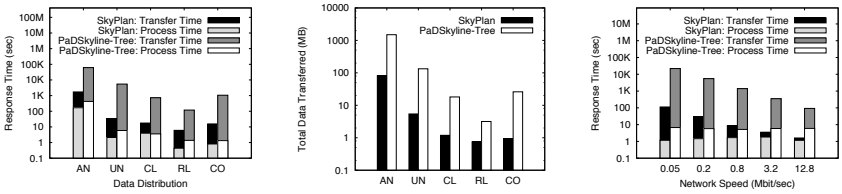
9.1 Comparative Performance Study

In Figures 7, 8 and 9 we present a thorough comparative study of SkyPlan against PaDSkyline for miscellaneous setups and different data distributions. We assess the performance of both approaches using the same filter strategy, in order to compare the produced execution plans directly, without interference from other factors. Notice that the y-axis is in logarithmic scale in all these charts.

First, we assess the performance of SkyPlan for uniform data distribution in Figure 7. In order to test the scalability of SkyPlan, we vary the dimensionality (Figure 7(a)), the number of points stored at each server (Figure 7(b)), and the number of servers (Figure 7(c)). In Figure 7(a), we compare the response time of SkyPlan against PaDSkyline. SkyPlan is 2-3 orders of magnitude better than PaDSkyline-Tree, and this gain is sustained with increased dimensionality. This result verifies that the execution plan deployed by SkyPlan improves the performance of query processing. The height of the generated plans are 6-11 in all setups thus verifying that it does not degenerate to the linear execution plan (figures are omitted due to lack of space). As expected, the response time required by PaDSkyline-Linear is even higher, which deems it impractical for distributed settings. Hence, it is omitted in the following charts.

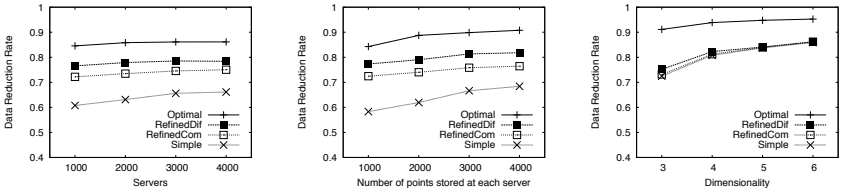
Then, in Figure 7(b), we depict the number of total transferred data by SkyPlan compared to PaDSkyline-Tree for varying the number of points per server. In all cases, SkyPlan transfers more than 1 order of magnitude fewer data. This indicates that SkyPlan defines the execution order of the queries in such a way that the amount of transferred data is reduced. In Figure 7(c), we study the response time for varying number of servers and we provide a cost breakdown analysis of the response time, in order to clearly illustrate the effect of networking and processing cost. The chart shows that the total gain in response time is both due to the reduction of transferred data (networking cost) as well as the reduction of processing time (plan generation, skyline computation and result merging). Moreover, notice that SkyPlan's processing cost for the generation of the execution plan is amortized over the cost of query execution. We also investigate the performance of SkyPlan against both variants of PaDSkyline for clustered data distribution in Figure 8. Again, we observe that SkyPlan shows consistently better performance than PaDSkyline for all evaluated metrics and for different setups.

Furthermore, in Figure 9, we evaluate the response time and total transferred data of SkyPlan and PaDSkyline for different data distributions. This experiment includes additionally correlated, anti-correlated and real data. We notice that SkyPlan achieves better processing and transfer time (Figure 9(a)) regardless of the data distribution. Even in the case of the demanding anti-correlated dataset, SkyPlan is almost 2 orders of magnitude better than PaDSkyline. In Figure 9(b), we evaluate the total amount of transferred data. Again SkyPlan is much more efficient (usually more than 1 order of magnitude) in transferred data in all setups. In Figure 9(c), we vary the network speed, in order to verify that the gain in transfer time of SkyPlan is sustained. Obviously both approaches present almost constant processing time for varying network speed. Also, as expected, the transfer time is reduced with higher network speed for both approaches. The important finding is that the improvement of SkyPlan in transfer time is sustained both for lower and higher transfer rates. This is a strong argument in favor of SkyPlan, as its performance gains are attained irrespective of the network speed.



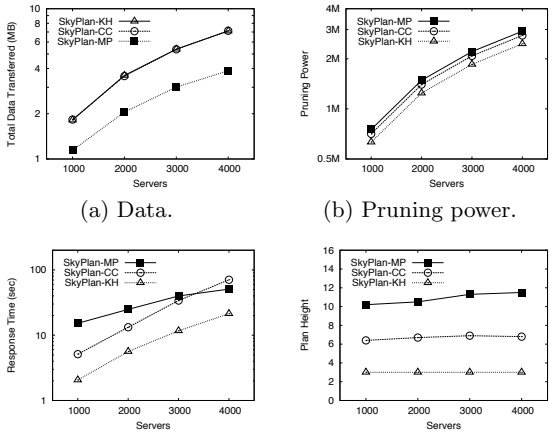
(a) Time vs. distribution. (b) Data vs. distribution. (c) Time vs. network speed.

Figure 9: Comparative performance for different data distributions and network speed (UN distribution).



(a) DRR for clustered data. (b) DRR for clustered data. (c) DRR for uniform data.

Figure 10: Data reduction rate (DRR) for different filter point strategies.



(a) Data. (b) Pruning power.

(c) Time. (d) Height.

Figure 11: Multi-objective execution plans for varying number of servers.

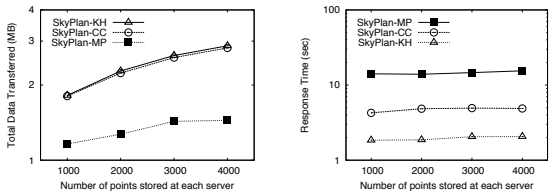
9.2 Effect of Filter Points

In the next series of experiments, we focus on the filter strategy and compare the different strategies using Data Reduction Rate (DRR). We also define a theoretical *optimal* filter strategy that is an upper bound of DRR that can be achieved. The optimal strategy assumes that the complete skyline SKY_i and the complete set of received filter points F_i are sent to the next servers S_j. In this case the filter points received by S_j are $F_j = SKY_i \bigcup F_i$. By transferring all these points, we capture the maximum pruning power possible. In order to define an optimal value for DRR, instead of using the $|F_j|$ during the DRR computation we use $min(k, |F_j|)$. Thus, the optimal approach transfers all available points, while paying only for k points as a cost. This optimal is not realistic nor feasible, but it still defines an upper bound for the DRR, thus establishing a ceiling for the performance of any algorithm.

Figure 10 presents the DRR for different filter strategies and various settings. In Figure 10(a) and Figure 10(b) we study the performance for clustered data distribution. Our strategy that sends the same filter points to all servers is denoted as *RefinedCom*, while *RefinedDif* denotes the case that different filter points are used for each server. We compare to the maxSum strategy proposed in [7], here denoted as *Simple*. Our strategies perform better than the competitor approach by 15%-33%. It is noteworthy that the RefinedDif strategy manages to significantly reduce the gap to optimal and to achieve gains in transferred data that are close to the theoretical optimal. Moreover, our strategies scale with the number of servers (Figures 10(a)) and the cardinality on each server (Figure 10(b)). In terms of comparison of our strategies, we observe that RefinedDif outperforms RefinedCom, because RefinedDif refines the filter points for each MBR individually, thus achieving better pruning power. For uniform data distribution, presented in Figure 10(c), all approaches are comparable which is expected as data objects are uniformly distributed in the data space. However, all filter strategies achieve high DRR values.

9.3 Multi-objective Execution Plans

In this section, we evaluate the performance of multi-objective execution plans, namely the k-hop execution plan (*SkyPlan-KH*) and the capacity constrained execution plan (*SkyPlan-CC*), compared to the maximum pruning execution plan (*SkyPlan-MP*). The maximum number of hops for the SkyPlan-KH is set to 3, while the maximum capacity for SkyPlan-CC is set to 20. In Figure 11, we compare the performance of the execution plans varying the number of servers for clustered data distribution. Figure 11(a) depicts the total transferred data for the different execution plans. SkyPlan-MP transfers fewer data than SkyPlan-KH and Skyplan-CC because its execution plan has the highest pruning power (Figure 11(b)). The high pruning power of the maximum pruning execution plan leads to filter points that prune more local skyline points. Consequently, the total transferred data is reduced. Figure 11(b) shows the total pruning power, i.e., the sum of edges' weights of the different execution plans. Clearly, there is a small loss in pruning



(a) Data. (b) Time.

Figure 12: Multi-objective execution plans for varying cardinality.

power for the multi-objective query plans caused by the imposed constraints. On the other hand, SkyPlan-KH presents the best performance in terms of response time, as depicted in Figure 11(c). The main reason is that the maximum number of hops required to process a skyline query is bounded by the fixed height of the execution plan. This leads to lower latency, thus reducing the response time. SkyPlan-CC generates balanced execution plans, which indirectly leads to plans with smaller height than SkyPlan-MP, because the capacity constrained execution plan avoids the existence of one long path. Therefore, SkyPlan-CC outperforms SkyPlan-MP in terms of response time for small number of servers, but for higher number of servers (> 3000) SkyPlan-MP performs better. The height of the different execution plans is depicted in Figure 11(d).

Figure 12 presents a comparative performance of different execution plans varying the number of points stored at each server for clustered data distribution. Figure 12(a) depicts the total amount of transferred data for each execution plan. SkyPlan-MP transfers around 50% fewer data than the other plans. This verifies that an execution plan with high pruning power reduces the amount of transferred data. In Figure 12(b), we evaluate the response time of the different execution plans. SkyPlan-KH achieves the smallest response time followed by Skyplan-CC. Nevertheless, the performance of all approaches is not influenced by the cardinality of the dataset, indicating the robustness of our framework when the size of the dataset increases.

10. CONCLUSIONS

In this paper, we address the problem of deriving efficient execution plans for distributed skyline computation. A distributed skyline query can be processed by evaluating multiple constrained skyline queries on different servers. We propose a novel framework, called SkyPlan, that maps the dependencies between the queries into a graph and generates cost-aware execution plans. Our main goal is to maximize the pruning power between consecutive queries, while at the same time increase parallelism, when no significant dependencies exist among queries. Moreover, SkyPlan supports multi-objective execution plans when additional constraints – other than maximizing the pruning power – need to be enforced. Based on the derived execution plan, we propose a distributed query execution mechanism that allows continuous refinement of the execution plan during in-network query processing. In our experimental evaluation, we show that SkyPlan outperforms the state-of-the-art algorithm.

11. REFERENCES

[1] A. Abdalla and N. Deo. Random-tree diameter and the diameter-constrained MST. *International Journal of Computer Mathematics*, 79(6):651–663, 2002.

[2] A. Amberg, W. Domschke, and S. Voß. Capacitated minimum spanning trees: algorithms using intelligent search. *Combinatorial Optimization:Theory and Practice*, 1:9–40, 1996.

[3] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *Proc. of EDBT*, pages 256–273, 2004.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of ICDE*, pages 421–430, 2001.

[5] L. Chen, B. Cui, H. Lu, L. Xu, and Q. Xu. iSky: Efficient and progressive skyline computing in a structured P2P network. In *Proc. of ICDCS*, pages 160–167, 2008.

[6] Y. Chu and T. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[7] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou. Parallel distributed processing of constrained skyline queries by filtering. In *Proc. of ICDE*, pages 546–555, 2008.

[8] J. Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240, 1967.

[9] D. Elias and M. Ferguson. Topological design of multipoint teleprocessing networks. *IEEE Trans. on Communications*, 22(11):1753–1762, 1974.

[10] K. Fotiadou and E. Pitoura. BITPEER: Continuous subspace skyline computation with distributed bitmap indexes. In *Proc. of DAMAP*, pages 35–42, 2008.

[11] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[12] L. Gouveia. Using the Miller-Tucker-Zemlin constraints to formulate a minimal spanning tree problem with hop constraints. *Computers & Operations Research*, 22(9):959–970, 1995.

[13] K. Hose, C. Lemke, and K.-U. Sattler. Processing relaxed skylines in PDMS using distributed data summaries. In *Proc. of CIKM*, pages 425–434, 2006.

[14] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in MANETs. In *Proc. of ICDE*, page 66, 2006.

[15] B. A. Julstrom. Greedy heuristics for the bounded diameter minimum spanning tree problem. *ACM Journal of Experimental Algorithmics*, 14:1.1–1.14, 2009.

[16] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. AGiDS: A grid-based strategy for distributed skyline query processing. In *Proc. of GLOBE*, 2009.

[17] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proc. of SIGMOD*, pages 227–238, 2008.

[18] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. SKYPEER: Efficient subspace skyline computation over distributed data. In *Proc. of ICDE*, pages 416–425, 2007.

[19] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. Efficient routing of subspace skyline queries over highly distributed data. *Trans. on Knowledge and Data Engineering (TKDE)*, 22(12):1694–1708, 2010.

[20] A. Vlachou and K. Nørvåg. Bandwidth-constrained distributed skyline computation. In *Proc. of MobiDE*, 2009.

[21] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *Proc. of ICDE*, pages 1126–1135, 2007.

[22] S. Wang, Q. H. Vu, B. C. Ooi, A. K. H. Tung, and L. Xu. Skyframe: A framework for skyline query processing in peer-to-peer systems. *VLDB Journal*, 18(1):345–362, 2009.

[23] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proc. of EDBT*, pages 112–130, 2006.

[24] L. Zhu, Y. Tao, and S. Zhou. Distributed skyline retrieval with low bandwidth consumption. *Trans. on Knowledge and Data Engineering (TKDE)*, 21(3):384–400, 2009.