Parallel Skyline Queries

Foto N. Afrati
National Technical University of Athens
Greece
afrati@softlab.ntua.gr

Paraschos Koutris
University of Washington
Seattle, WA
pkoutris@cs.washington.edu

Dan Suciu
University of Washington
Seattle, WA
suciu@cs.washington.edu

Jeffrey D. Ullman
Stanford University
USA
ullman@infolab.stanford.edu

ABSTRACT

In this paper, we design and analyze parallel algorithms for skyline queries. The skyline of a multidimensional set consists of the points for which no other point exists that is at least as good along every dimension. As a framework for parallel computation, we use both the MP model proposed in (Koutris and Suciu, PODS 2011), which requires that the data is perfectly load-balanced, and a variation of the model in (Afrati and Ullman, EDBT 2010), the GMP model, which demands weaker load balancing constraints. In addition to load balancing, we want to minimize the number of blocking steps, where all processors must wait and synchronize. We propose a 2-step algorithm in the MP model for any dimension of the dataset, as well a 1-step algorithm for the case of 2 and 3 dimensions. Moreover, we present a 1-step algorithm in the GMP model for any number of dimensions.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Parallel Databases

General Terms

Algorithms, Theory

Keywords

Database Theory, Skyline Queries, Parallel Computation

1. INTRODUCTION

The availability of large and cheap server clusters nowadays has generated a lot of interest in large scale data analytics. The parallel computation model supported by today's clusters of commodity servers is usually some variation of the *map-reduce* model of computation, which was introduced in [6]. In this paper we present new algorithms for computing skyline queries on server clusters, with theoretical guarantees. To phrase and prove these theoretical guarantees, we use a formal model of parallel computation derived from [1] and [13].

Skyline Queries. Skyline queries were introduced in [3], in a context where the database is a collection of objects that can be rated by multiple criteria. For example, a restaurant can be rated based on its price, quality of service and quality of food. In this case, a skyline query will return the set of all the restaurants such that no other restaurant is at least as good in all three criteria and better in at least one.

Formally, given a d-dimensional set [1] $R(X_1, \dots, X_d)$ with n data items, the domination relationship $\preceq$ and the skyline, denoted by $S(R)$, are defined as follows:

Definition 1.1 (Domination). *A point $\mathbf{x} \in R$ dominates $\mathbf{x}' \in R$, which is denoted by $\mathbf{x} \preceq \mathbf{x}'$, if for every dimension $k = 1, 2, \dots, d$ we have*[2] $\mathbf{x}_k \leq \mathbf{x}'_k$.

Definition 1.2 (Skyline). *The skyline $S(R)$ of a d-dimensional set R consists of all maximal elements for the domination relationship, i.e.*

$$S(R) = \{\mathbf{x} \in R \mid \forall \mathbf{y} \in R,\ \textit{if } \mathbf{y} \preceq \mathbf{x} \textit{ then } \mathbf{y} = \mathbf{x}\}$$

Parallel Model of Computation. A parallel algorithm on a modern server cluster runs on P servers connected by a fast network. The data is initially distributed evenly among the servers, such that each server holds $O(n/P)$ data, where n is the number of data items. The computation proceeds in rounds, where each round consists of local computation, followed by a global data exchange. For example, a map-reduce job consists of computation (map), data exchange and computation (reduce).

There are two main complexity parameters in a parallel algorithm. The first is the number of communication rounds, also called *synchronization complexity* [10]. Each synchronization step adds a significant amount to the algorithm's running time, because all servers have to wait for the slowest worker. For that reason, the number of synchronization steps is an important complexity parameter. The second parameter is the *maximum load per server*. If a server must process more data items than others, then it will slow down

[1]Throughout this paper, we will assume set (and not bag) semantics.
[2]Note that, if $\mathbf{x}$ and $\mathbf{x}'$ are distinct, then for at least one dimension k, $\mathbf{x}_k < \mathbf{x}'_k$.

the entire computation. Also, it may force the use of disk at that processor, rather than using only main memory. To achieve a low server load, a parallel algorithm needs to both divide evenly the data among the servers, and avoid replicating the same data item to multiple servers. Keeping the load per server low indirectly benefits another important parameter, the total amount of data exchanged by the servers, since the amount of data being exchanged is upper bounded by the total load at all servers.

There is often a tradeoff between the number of synchronization steps and the server load: adding more communication rounds may result in a reduction of the maximum sever load. At an extreme, any sequential algorithm can be "parallelized" using a single synchronization step in a very naïve way, by sending all the data to the first server and then solving the problem locally. However, this increases the maximum server load from the optimal $O(n/P)$ to n.

To present our algorithms for computing the skyline, we need a formal theoretical model for parallel computation. Several models have been proposed in the literature for analyzing parallel algorithms on server clusters [7, 1, 13, 11]. In this paper, we study algorithms based on the frameworks of [1] and [13].

The Massively Parallel (MP) model [13] has two distinct features. First, it requires that the maximum load per server is $O(n/P)$, for any input database. This constraint implies that the data cannot be replicated by more than a constant factor on average; therefore, the single parameter of interest is the number of communication rounds. This is essentially the same requirement as in the Coarse Grained Multicomputer model introduced in [7]. Second, the MP model introduces a *broadcast phase*, during which the servers can exchange a limited amount of data. We should emphasize here that the broadcast phase is not counted as a synchronization step. Typically, the broadcast phase is used to detect skewed elements or gather other information about the data. For example, in Pig Latin's *skew-join*, the frequent items are computed first, then treated separately during the join [8]. It was shown in [13] that a broadcast phase is necessary in order to guarantee load balancing, even in the case of simple queries, for example a semi-join query.

An alternative model is described by [1], where the authors allow some data to be replicated more than a constant number of times. An algorithm in that model would use information about the size of the tables and decide which table(s) to replicate. A simple example is the *broadcast join*, which broadcasts the smaller table to all servers: if the size of the smaller table is less than $1/P$ the size of the larger table, then the maximum load per server is $< 2n/P$. It was shown in [1] that every Conjunctive Query can be computed using a single synchronization step in this model: if applied to an arbitrary database, this algorithm results in an average server load $O(n/P^{\varepsilon})$, for some[3] $\varepsilon > 0$; in other words, the entire data is replicated by an average factor of $P^{1-\varepsilon}$. This upper bound assumes that the data is skew-free. The one-step algorithm for an arbitrary Conjunctive Query is a rather surprising result, and is also an excellent illustration of the tradeoff between load balancing and the number of computation steps: some Conjunctive Queries cannot be computed in one parallel step if one requires a maximum load balance of $O(n/P)$ data items per server [13].

[3]If the Conjunctive Query has k variables, then ε is at most $1/k$.

Our contribution. In this paper, we propose three new algorithms for computing skyline queries on server clusters. We use both the MP model of [13], and a weakly load-balanced variation of the model of [1], which we call here the GMP model.

Let n denote the size of the input relation R, d the dimension of the data, and P the number of servers available. We present three algorithms for computing the skyline $S(R)$.

- The first algorithm (algorithm 1) uses two communication steps and is perfectly load-balanced: more precisely, the maximum server load is $O(dn/P)$. This algorithm is described in Subsection 4.1.
- The second algorithm (algorithm 2) uses only one communication step, but the maximum server load is increased to $O(dn/P^{1/(d-1)})$. In other words, the entire data is replicated on average by a factor of $P^{(d-2)/(d-1)}$, saving one communication step over the previous algorithm. We also describe conditions under which the load of this algorithm drops to $O(n/P)$. This algorithm is described in Subsection 4.2.
- The third algorithm (algorithm 3) computes the skyline for a database of dimension $d = 3$. It has a single communication step and is perfectly load-balanced: more precisely, the maximum sever load is $O(n/P)$. This algorithm is described in Subsection 4.3.

All three algorithms are based on the technique of *grid-based partitioning*. For each of the d dimensions, we partition the data into P *buckets* of roughly equal size $O(n/P)$: the $d \cdot P$ partition points (P points for each axis) are computed by the servers by broadcasting $d \cdot P^2$ data items. Each data point $\mathbf{x} \in R$ belongs to exactly d buckets, one along each dimension.

At a high level, our first algorithm partitions the data into these buckets (replicating each data item d times). Server i is responsible for the i-th bucket in each dimension, hence it holds $O(dn/P)$ data. Next, each server computes the skyline locally in each bucket. We show that a point $\mathbf{x}$ is in the skyline iff it is in the local skyline of each of its d buckets. However, these d copies of the data point may reside on different servers; hence, a second communication step is needed to bring all copies of a point to a common server, and compute the skyline $S(R)$. As we will show in Subsection 4.1, our first algorithm develops this basic idea in several ways. It is, to the best of our knowledge, the first provably load-balanced algorithm that computes the skyline for d-dimensional data, for arbitrary d, using only two communication steps.

The second algorithm saves one synchronization step over the first algorithm, at the cost of an increased amount of data replication. It starts similarly, by partitioning each dimension into buckets, but uses only M buckets, where $M \ll P$. We use the term *cell* to denote the intersection of d buckets (one in each dimension). Thus, there are M^d cells, and each data point belongs to exactly one cell. After computing the bucket boundaries, the servers compute the set of nonempty cells by broadcasting M^{d+1} bits. Then, each server filters out the cells that are strictly dominated by another nonempty cell. We prove that only $O(M^{d-1})$ cells remain after this filter operation. Furthermore, only the data points in these remaining cells (and the cells dominating them) need to be inspected in order to compute the skyline. Our algorithm follows by choosing M such that $M^{d-1} = P$. The algorithm, in essence, replicates data items by a factor of $P^{(d-2)/(d-1)}$ in order to save one communication step. Notice that for 2-dimensional data, this factor is 1, and therefore the maximum load per server is $O(n/P)$.

Our third algorithm improves the load balancing guarantee in the special case of 3-dimensional data: this algorithm runs in one parallel step, and has a maximum load of $O(n/P)$ per server. It is, to the best of our knowledge, the first provably load-balanced algorithm that computes the skyline for 3-dimensional data using a single communication step.

We leave open the question whether there exists a one-step algorithm with a $O(n/P)$ maximum load for arbitrary d-dimensional data.

Related Work. There exists a rich literature related to the computation of *skyline queries* (the term used in databases), or of the *maximal vector problem* (the term used in computational geometry). The theoretical time complexity of the problem was first studied in [14]. Other papers followed, including [21] and [16], that applied divide-and-conquer techniques and matrix multiplication respectively. After the introduction of the skyline operator in the database community by [3], several efficient algorithms were developed [4, 9, 17, 15].

In recent years, there has been an increasing interest in parallelizing skyline queries for distributed and parallel environments. All approaches share the idea of partitioning the set of points, processing locally the partitions in parallel, and finally combining their results. Their difference resides in the partitioning schemes of the data.

The most common approach is grid-based partitioning [24, 20, 23, 7]. The idea is to create a grid on the data, such that each cell of the grid has roughly the same amount of data. The local skyline of each cell is computed in parallel, and the final result is obtained by merging the local skylines. Another partitioning technique applied in [5] is random partitioning. Using randomness guarantees that the points will be distributed in a uniform fashion among the partitions; however, many points may belong in the local skyline of the partition but not in the final skyline.

Recent work focuses on an angle-based space partitioning scheme, first proposed by [22]. The algorithm transforms the points using hyperspherical coordinates before partitioning into a grid, a technique that alleviates the problem of computing the local skylines of cells that do not participate in the global skyline. In [12], the authors partition the space using hyperplane projections, an approach close to angle-based partitioning. Their algorithm also uses a preprocessing step to quickly filter out a part of the dominated points, as well as a more efficient merging technique. In [18] skyline computation is parallelized for multicore architectures, under the assumption that the participating cores share everything and communicate by updating main memory. The technique applied is a divide-and-conquer strategy combined with an iterative sequential algorithm.

All aforementioned techniques divide the space into disjoint cells, and then merge the cells recursively by applying repeatedly the identity $S(R_1 \cup R_2) = S(S(R_1) \cup S(R_2))$. In order to compute the skyline of $R_1 \cup R_2$, one first computes in parallel the skylines $S(R_1)$ and $S(R_2)$, which are hopefully much smaller than R_1, R_2, then merges the result, by computing the skyline of $S(R_1) \cup S(R_2)$. This is a recursive divide-and-conquer algorithm, and requires $\log P$ communication steps, which is far worse than the one or two communication steps achieved by our algorithms. In order to reduce the number of communication steps, we apply a different principle than recursive divide and conquer. Our technique uses overlapping buckets (along each dimension): if R, T are two overlapping buckets, and $C = R \cap T$ is the cell of their intersection, then the skyline points in C are precisely those points that belong both to the skyline of R and to the skyline of T, that is, $C \cap S(R \cup T) = S(R) \cap S(T)$. There is no need for recursive merging, and this allows us to reduce the parallel running time to one or two steps. To the best of our knowledge, the three algorithms we present here are the first that have $O(1)$ communication steps (in fact, only one or two steps, respectively), and are guaranteed load-balanced.

Closest to our results is the work of [7], which describes a parallel algorithm for the skyline over 3-dimensional data; the skyline problem is called there 3D-Maxima. Their algorithm uses similar ingredients to ours: it partitions the three dimensional data into equal buckets along the X-dimension, and into equal buckets along the Z-dimension (it does not partition it along the Y-dimension). It starts by computing the skyline in the X-buckets, then passes them to the Z-buckets, which compute their skylines and intersect them with those from the X-buckets. To avoid the third intersection step (with the Y-buckets) the authors make a clever use of a 2-dimensional skyline that they compute during the first step. When cast into our model, their algorithm uses two synchronization steps, and is perfectly load-balanced, hence it is similar to our first algorithm, but does not generalize beyond 3 dimensions. In constrast, for the special case of 3 dimensions our third algorithm computes the skyline in only one step: to the best of our knowledge this is the first algorithm to achieve that.

Outline. The paper is organized as follows. Section 2 reviews the two models of parallel computation in detail. Next, in Section 3 we present the preprocessing steps that are common to all the algorithms, and also provide some useful tools for analyzing our algorithms. In Section 4 we describe the three algorithms for skyline queries in full detail and also analyze their performance. Finally, we conclude in Section 5.

2. PARALLEL COMPUTATION MODELS

We review here the Massively Parallel (MP) model of computation from [13] and describe its generalization (GMP) by adopting ideas from [1]. We denote with n the size of the data: in the case of a skyline query, n is the size of a d-dimensional set $R(X_1, \ldots, X_d)$. Computation in the MP model is performed by P servers, having unbounded local memory, and connected by a network. Initially, the input data is uniformly partitioned across the P servers, such that each server holds n/P data items. In the case of a skyline query, we denote R_s the fragment of R stored at server s, for $s = 1, \ldots, P$; thus $\bigcup_s R_s = R$, $|R_1| = \ldots = |R_P| = n/P$.

The computation consists of a sequence of global parallel steps, each consisting of three phases:

Broadcast Phase: The P servers exchange a small amount of global data, called *broadcast data*, and perform some computation on their local data and the broadcast data.

Communication Phase: The P servers exchange data.

Computation Phase: The P servers perform some local computation on their local data.

The main parameter of the model is the number of parallel steps. For example, an algorithm that can solve the problem in one parallel step taking time T is considered superior to an algorithm that can solve the problem in two parallel steps, each taking time $T/2$.

There are two constraints imposed in the MP model. First, the amount of data exchanged during a broadcast step should depend[4] only on P, and be independent on n. Second, the model imposes a strict load balance requirement: denoting n_s the number of data items held by server s throughout the algorithm, it is required that $\max_{s=1,P} n_s = O(n/P)$. For a randomized algorithm, the load balancing requirement becomes $\mathbb{E}[\max_{s=1,P} n_s] = O(n/P)$, where the expectation is taken over the random choices of the algorithm. Thus, the MP model requires all servers to be perfectly load-balanced. Notice that if we allowed the server load to increase to $\max_{s=1,P} n_s = O(n)$, then it is trivial to design a one-step algorithm for computing the skyline: simply send all the data to sever $s = 1$, and compute the skyline locally there.

In this paper we also consider the following generalization, called the GMP model, where the load balance requirement is relaxed to $\max_{s=1,P} n_s = O(n/P^{\varepsilon})$, for some $\varepsilon > 0$; note that if we allowed $\varepsilon = 0$ then the maximum load balance would increase to $O(n)$, which we want to forbid; hence we are only interested in the case $\varepsilon > 0$. Thus, while in the MP model the data may be replicated by at most a constant average factor, in the GMP model the data may be replicated by an average factor of $O(P^{1-\varepsilon})$. In practice, a major concern in designing parallel algorithms (e.g. in the Map-Reduce framework) is reducing the amount of data exchange. In our case, it follows that the total amount of data exchanged in one parallel step is $O(n)$ (in the MP model) and $O(nP^{1-\varepsilon})$ (in the GMP model).

In this paper, we study algorithms for computing the skyline in the MP and the GMP model, and we present algorithms that run in one, or two parallel steps.

3. PREPROCESSING

In this section, we present the preprocessing steps that are common to all three of our algorithms. Furthermore, we provide several definitions and propositions, which will prove useful in Section 4.

3.1 Bucketizing

A basic primitive of our techniques is the partitioning of the d-dimensional relation R into M buckets across some dimension k, such that each partition contains approximately the same number of points, i.e. $O(n/M)$ points. The value of M will be chosen later[5], and depends only on the number of servers P: it is independent on the number of data items n. As a consequence, we will show that this partition can be performed using one broadcast step, by broadcasting $d \cdot P \cdot (M + 1)$ data items, which is independent of n.

Assume for now that all the n data items in our set R have distinct coordinates: in other words, for any $\mathbf{x}, \mathbf{y} \in R$, $\mathbf{x} \neq \mathbf{y}$, and for any dimension $k = 1, \ldots, d$, $\mathbf{x}_k \neq \mathbf{y}_k$; we show how to drop this assumption in the next subsection.

Fix a dimension $k = 1, \ldots, d$. We will choose $(M + 1)$ partition points

$$-\infty = b_k^0 < b_k^1 < \cdots < b_k^{M-1} < b_k^M = +\infty$$

and for each $i = 1, \ldots, M$, we define the *bucket* B_k^i as

$$B_k^i = \{\mathbf{x} \in R \mid b_k^{i-1} \leq \mathbf{x}_k < b_k^i\}$$

Our goal is to compute the partition points b_k^i such that $\forall i, k,\ |B_k^i| = O(n/M)$. This is done during a broadcast phase, and we call this algorithm BUCKETIZE.

The algorithm works as follows. For each dimension k, every server s bucketizes its local fragment R_s along dimension k, by computing partition points $-\infty = y_{s,k}^0 < y_{s,k}^1 < \cdots < y_{s,k}^M = +\infty$, such that each local bucket $B_{s,k}^i = \{\mathbf{x} \in R_s \mid y_{s,k}^{i-1} \leq \mathbf{x}_k < y_{s,k}^i\}$ has $|R_s|/M = n/(MP)$ data points. This can be done, for example, by sorting R_s on dimension k and choosing every (n/M)-th point.

Next, each server broadcasts the local partition points $y_{s,k}^i$: the total number of values broadcast is thus $dP(M+1)$. Once the data is broadcast to all servers, each server does the following for each dimension k. It first sorts the points $y_{s,k}^i$ in increasing order, say $-\infty = z^0 < \cdots < z^{P(M-1)+1} = +\infty$. Notice that, for any dimension k, there are no duplicates among the values $y_{s,k}^i$, except for $-\infty, +\infty$ which occur at each server s. Then, it selects every P-th value: $b_k^i = z^{iP}$ for $i = 0, \ldots, M-1$, plus $b_k^M = +\infty$ (notice that also $b_k^0 = -\infty$). The following proposition establishes the correctness of the algorithm.

PROPOSITION 3.1. *Algorithm* BUCKETIZE *computes $M + 1$ buckets for each dimension, s.t. each bucket contains $O(n/M)$ points. It broadcasts a total of $dP(M + 1)$ data items.*

PROOF. Let us consider a bucket B_k^i. This bucket includes all points $\mathbf{x} \in R$ such that $z_k^{iP} \leq \mathbf{x}_k < z^{(i+1)P}$. There are a total of $P + 1$ partition points between z_k^{iP} and $z_k^{(i+1)P}$, including z^{iP} and $z^{(i+1)P}$. Let $n_{s,k}$ be the number of local partition points that server s contributes to these points (in other words, $n_{s,k} = |\{j \mid z_k^{iP} \leq y_{s,k}^j \leq z_k^{(i+1)P}\}|$). Then, it is easy to see that B_k^i will contain points from at most $n_{s,k} + 1$ local buckets of server s, and for each such bucket we have $|B_{s,k}^j| \leq n/(MP)$. Moreover, it holds that $\sum_s n_{s,k} = P + 1$.

Hence, the total number of points contained in B_k^i is

$$\begin{aligned} |B_k^i| &\leq \sum_{s=1}^{P} (n_{s,k} + 1) \cdot n/(MP) \\ &= (2P + 1) \cdot n/(M \cdot P) \approx 2n/M \end{aligned}$$

This concludes the proof. □

3.2 Handling Equal Coordinates

If two distinct points $\mathbf{x}, \mathbf{y} \in R$ have a common dimension, say $\mathbf{x}_k = \mathbf{y}_k$, then the bucketization algorithm must break ties. In the extreme case, all n points have the same k-th coordinate, and, without any changes, the algorithm BUCKETIZE would fail to partition the points into k equal buckets along the k-th dimension. We show here how to break ties by using the other dimensions. Note that if two points agree

[4]In [13], the size of the broadcast data was required to be $O(n^{\varepsilon})$, for some $\varepsilon < 1$. In this paper we impose a stricter bound, by requiring it to be independent on n.

[5]It is either P, $P \log P$ or $P^{1/(d-1)}$.

on *all* dimensions, i.e. $\mathbf{x}_1 = \mathbf{y}_1, \ldots, \mathbf{x}_d = \mathbf{y}_d$, then the two points must be equal, since R is a set.

More precisely, we will describe a transformation of the points to points with distinct values for every dimension. The mapping t we use is as follows:

$$t(p_1, \ldots, p_d) = $$
$$((p_1, p_2, \ldots, p_d), (p_2, \ldots, p_d, p_1), \ldots, (p_d, p_1, \ldots, p_{d-1}))$$

We now introduce a new comparison operator ($\trianglelefteq$) for the transformed points. For the tuples $\mathbf{p} = (p_1, \ldots, p_d)$ and $\mathbf{q} = (q_1, \ldots, q_d)$, we define that $\mathbf{p} \trianglelefteq \mathbf{q}$ if either $\mathbf{p} = \mathbf{q}$ or there exists some index i such that $p_i < q_i$ and for every $j < i$ we have that $p_j = q_j$. If $\mathbf{p} \neq \mathbf{q}$ and $\mathbf{p} \trianglelefteq \mathbf{q}$, we also write that $\mathbf{p} \triangleleft \mathbf{q}$. This is a standard lexicographic order, and it is known to be a total order. Given a set R of d-dimensional data points, we denote by $R^t = \{t(\mathbf{x}) \mid \mathbf{x} \in R\}$ the set of transformed points.

The new set R^t is just a regular d-dimensional set, where the coordinates are ordered by the relation $\trianglelefteq$ rather than the natural order relation $\leq$. In particular, Definition 1.1 gives the domination relationship between the transformed points: $t(\mathbf{x}) \preceq t(\mathbf{y})$ iff for all $k = 1, \ldots, d$, $(t(\mathbf{x}))_k \trianglelefteq (t(\mathbf{y}))_k$. We will show that (1) the mapping t defines a one-to-one mapping between the skyline $S(R)$ and the skyline $S(R^t)$, and (2) no two points in R^t agree on any coordinate k (thus satisfying our assumption in Subsection 3.1).

Proposition 3.2. *$\mathbf{x} \preceq \mathbf{y}$ if and only if $t(\mathbf{x}) \preceq t(\mathbf{y})$.*

The proposition immediately implies that $\mathbf{x} = \mathbf{y}$ if and only if $t(\mathbf{x}) = t(\mathbf{y})$. In particular, t is an injective function.

Proof. Assume first that $\mathbf{x} \preceq \mathbf{y}$. Then, for every k, $x_k \leq y_k$. We will show that $(t(\mathbf{x}))_k = (x_k, \ldots, x_d, x_1, \ldots, x_{k-1}) \trianglelefteq (y_k, \ldots, y_d, y_1, \ldots, y_{k-1}) = (t(\mathbf{y}))_k$. If $\mathbf{x} = \mathbf{y}$ then obviously $t(\mathbf{x}) \trianglelefteq t(\mathbf{y})$, so assume that $\mathbf{x} \neq \mathbf{y}$. When ordering the indices as $k, \ldots, d, 1, \ldots, k-1$, consider the first index i such that $x_i \neq y_i$. Since $\mathbf{x} \preceq \mathbf{y}$, we must have $x_i < y_i$; moreover, by assumption, for all indices j preceding i in the order $k, \ldots, d, 1, \ldots, k-1$, we have $x_j = y_j$. Hence, $(t(\mathbf{x}))_k \trianglelefteq (t(\mathbf{y}))_k$. Since this holds for every k, we have $t(\mathbf{x}) \preceq t(\mathbf{y})$.

For the other direction, let $t(\mathbf{x}) \preceq t(\mathbf{y})$. Since for every k, we have that $(t(\mathbf{x}))_k \trianglelefteq (t(\mathbf{y}))_k$, it follows that $x_k \leq y_k$, and, hence, $\mathbf{x} \preceq \mathbf{y}$. $\square$

This implies immediately:

Corollary 3.3. *Let $R^t = \{t(\mathbf{x}) \mid \mathbf{x} \in R\}$. Then the transformation t is a one to one mapping between the two skyline sets $S(R)$ and $S(R^t)$.*

Hence, instead of computing the skyline of R, we can compute the skyline of the transformed set R^t. Then, $S(R)$ will be given by inversing the transformation, which is easily computed since t is one-to-one.

It remains to prove that the transformed points do not share any values for the same coordinate.

Proposition 3.4. *If $t(\mathbf{x}) \neq t(\mathbf{y})$, then for every $k = 1, d$, we have that $(t(\mathbf{x}))_k \neq (t(\mathbf{y}))_k$.*

Proof. For the sake of contradiction, assume that there exists some k such that $(t(\mathbf{x}))_k = (t(\mathbf{y}))_k$. However, this means that for every j it holds that $\mathbf{x}_j = \mathbf{y}_j$, which implies that $\mathbf{x} = \mathbf{y}$ and thus $t(\mathbf{x}) = t(\mathbf{y})$, a contradiction. $\square$

Finally, notice that we do not need to actually compute the transformation; it suffices to directly use the new comparison operator defined on $t(R)$. Hence, this technique does not increase the amount of data communicated.

In the rest of the paper we will assume that the transformation t has been applied to the data set R, and, therefore, all points in R have distinct coordinates.

3.3 The Relaxed Skyline of Cells

After having bucketized R across every dimension, we define the grid-based partitioning of the data into cells. We use the standard notation $[M] = \{1, 2, \ldots, M\}$.

Definition 3.5. *A cell $B(\mathbf{i})$, for any $\mathbf{i} = (i_1, \ldots, i_d) \in [M]^d$ is the set*

$$B(\mathbf{i}) = \bigcap_{j=1}^{d} B_j^{i_j}$$

The cells belong to a d-dimensional discrete grid with M^d points. Each bucket B_k^i corresponds to the hyperplane $i_k = i$ in this space; hence, we will refer interchangeably to buckets as hyperplanes. Moreover, for $\mathbf{i} \in [M]^d$, we interchangeably refer to $B(\mathbf{i})$ or $\mathbf{i}$ as a cell.

The following proposition follows directly from the definition of a cell.

Proposition 3.6. *Cells have the following properties.*

1. *Each cell holds $O(n/M)$ data.*
2. *Each point $\mathbf{x} \in R$ belongs to exactly one cell.*

We note that the first property above is a weak bound on the size of a cell. Since there are M^d cells, one would wish that $|B(\mathbf{i})| = O(n/M^d)$; instead, only the weaker property $|B(\mathbf{i})| = O(n/M)$ holds, and the bound is in fact tight. For example, the database may consist of n points on the diagonal, i.e. each point is of the form $(x, x, \ldots, x)$. In this case, the only nonempty cells are the diagonal cells $B(i, i, \ldots, i)$, where $i = 1, \ldots, M$, and each diagonal cell contains n/M elements. Once the algorithm Bucketize computes the bucket boundaries, each server knows the identity of each cell $B(\mathbf{i})$ (but not its data points).

Let us define the set of the non-empty cells as

$$J = \{\mathbf{i} \in [M]^d \mid B(\mathbf{i}) \neq \emptyset\}$$

Once the servers know the identity of all cells, they compute the set J: this requires broadcasting at most PM^d data items (each server locally checks whether each cell is empty and then broadcasts this information).

All three of our algorithms rely on computing the skyline locally in each cell. However, not all cells contain skyline points, as can be seen in Figure 1: for example, the cell (5,5) cannot contain any skyline points, because the entire cell is strictly dominated by, say, the cell (4,3). Our next task is to select only those cells that *may* contain skyline points. It turns out that this set can also be described as a kind of skyline, but in the space of cells rather than the space of data points, and by replacing the domination relation with strict domination. We give the formal definitions next.

Definition 3.7 (Strict Domination). *A point $\mathbf{i} \in J$ strictly dominates $\mathbf{j} \in J$, in notation $\mathbf{i} \prec \mathbf{j}$, if for every dimension $k = 1, 2, \ldots, d$ we have $\mathbf{i}_k < \mathbf{j}_k$.*

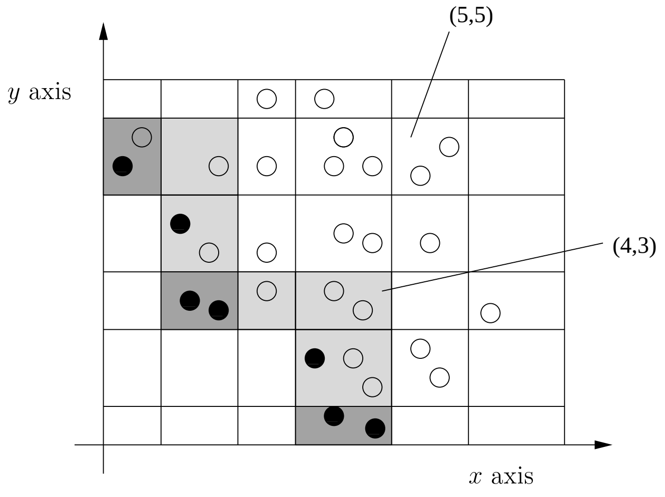

Figure 1: The grid-based partition into cells, along with the skyline (only dark grey) and relaxed skyline (light and dark grey) of the cells. The points painted black are the points of the global skyline of this instance.

Definition 3.8 (Relaxed Skyline). *The* relaxed skyline *(R-skyline) of a set J, denoted by $S_r(J)$, is the set of points of J that are not* strictly dominated *by some other point:*

$$S_r(J) = \{\mathbf{i} \mid \mathbf{i} \in J, \neg\exists \mathbf{j} \in J.(\mathbf{j} \prec \mathbf{i})\}$$

Notice that strict domination implies domination: $\mathbf{i} \prec \mathbf{j} \Rightarrow \mathbf{i} \preceq \mathbf{j}$. Therefore, the skyline is always a subset of the relaxed skyline, that is $S(J) \subseteq S_r(J)$. Figure 1 illustrates the skyline (dark grey) and the relaxed skyline (both light and dark grey) for the cells of a 2-dimensional data set.

The following two facts are easy to check. If $\mathbf{x} \in B(\mathbf{i})$ and $\mathbf{y} \in B(\mathbf{j})$, then (1) $\mathbf{x} \preceq \mathbf{y}$ implies $\mathbf{i} \preceq \mathbf{j}$ and (2) $\mathbf{i} \prec \mathbf{j}$ implies $\mathbf{x} \preceq \mathbf{y}$; however $\mathbf{i} \preceq \mathbf{j}$ does not imply in general $\mathbf{x} \preceq \mathbf{y}$. This explains our interest in the strict domination between cells, and by extension, in the relaxed skyline $S_r(J)$.

Once the P servers have computed J, each server computes the relaxed skyline $S_r(J)$. The following two lemmas show that, in order to compute the skyline query $S(R)$ we only need to know the points in the cells belonging to $S_r(J)$. Thus, these will be the only cells considered by our algorithms.

Lemma 3.9. *Consider a point $\mathbf{x}$ in a cell $\mathbf{i}$, $\mathbf{x} \in B(\mathbf{i})$. If $\mathbf{x} \in S(R)$, then $\mathbf{i} \in S_r(J)$.*

Proof. Suppose the contrary, $\mathbf{i} \notin S_r(J)$. Then, $\mathbf{i}$ is strictly dominated by some other point $\mathbf{j} \in J$, i.e. $\mathbf{j} \prec \mathbf{i}$. Consider any point $\mathbf{y} \in B(\mathbf{j})$ (such a point exists since all cells in J are nonempty). Clearly $\mathbf{y} \neq \mathbf{x}$, since $\mathbf{y}, \mathbf{x}$ belong to distinct cells, and we have argued earlier that $\mathbf{j} \prec \mathbf{i}$ implies $\mathbf{y} \preceq \mathbf{x}$. This contradicts the fact that $\mathbf{x} \in S(R)$. □

The lemma says that all the answers to our skyline query are in the cells $\mathbf{i} \in S_r(J)$: we need not look further. But we still need to check if a point $\mathbf{x} \in B(\mathbf{i})$ is a skyline point, by comparing it with other points $\mathbf{y}$. The next lemma says that $\mathbf{y}$, too, can be restricted to points belonging only in cells of the relaxed skyline.

Lemma 3.10. *Consider a data point $\mathbf{x}$ in a cell $\mathbf{i}$: $\mathbf{x} \in B(\mathbf{i})$. Suppose that $\mathbf{x} \notin S(R)$. Then there exists an R-skyline cell $\mathbf{j} \in S_r(J)$ and a point $\mathbf{y} \in B(\mathbf{j})$ such that $\mathbf{y} \preceq \mathbf{x}$.*

Proof. If $\mathbf{x} \notin S(R)$, then by definition there exists $\mathbf{y} \neq \mathbf{x}$ s.t. $\mathbf{y} \preceq \mathbf{x}$. It is easy to see that we can choose $\mathbf{y}$ to be a skyline point, $\mathbf{y} \in S(R)$. Then, by Lemma 3.9, the cell $\mathbf{j}$ of $\mathbf{y}$ is an R-skyline cell, $\mathbf{j} \in S_r(J)$, which proves our claim. □

By our discussion above, we also have $\mathbf{j} \preceq \mathbf{i}$. Thus, in order to compute the skyline points in the cell $B(\mathbf{i})$, we only need to inspect the data points in the cells $B(\mathbf{j})$ such that $\mathbf{j} \in S_r(J)$ and $\mathbf{j} \preceq \mathbf{i}$. For any $\mathbf{i} \in S_r(J)$, let us define

$$N(\mathbf{i}) = \{\mathbf{j} \in S_r(J) \mid \mathbf{j} \preceq \mathbf{i}\}$$

Notice that for each $\mathbf{j} \in N(\mathbf{i})$ there exists a dimension k s.t. $j_k = i_k$: indeed, otherwise we have $j_k < i_k$ for each dimension, which implies $\mathbf{j} \prec \mathbf{i}$, contradicting the fact that $\mathbf{i}$ is in the relaxed skyline.

Our last technical result in this subsection computes the total number of data points in all cells $B(\mathbf{j})$, where $\mathbf{j} \in N(\mathbf{i})$.

Lemma 3.11. *Fix a cell $\mathbf{i} \in S_r(J)$. The total number of data points in all cells $B(\mathbf{j})$, for $\mathbf{j} \in N(\mathbf{i})$, is $O(n/M)$.*

Proof. Each cell $\mathbf{j} \in N(\mathbf{i})$ shares a common hyperplane k with $\mathbf{i}$: in other words, $j_k = i_k$ for some k. It follows that the cell $B(\mathbf{j})$ is a subset of the bucket $B_k^{i_k}$. Hence, the union of all cells $B(\mathbf{j})$ is included in the union of the d buckets (hyperplanes) that contain the cell, which, together, have $d \cdot O(n/M) = O(n/M)$ points. □

Thus, a naïve way to compute the skyline $S(R)$ is to send a copy of all the data points in all cells $B(\mathbf{j})$, where $\mathbf{j} \in N(\mathbf{i})$, to the cell $B(\mathbf{i})$, then check locally which points $\mathbf{x} \in B(\mathbf{i})$ remain not dominated. The lemma shows, quite surprisingly, that only $O(n/M)$ data items need to be sent to the cell $B(\mathbf{i})$, which means that by choosing $M = P$, this computation can be done by one server. Unfortunately, as we show next, the number of cells $\mathbf{i} \in S_r(J)$ may be too large to make this naïve algorithm work.

3.4 The Size of the Relaxed Skyline of Cells

In this subsection, we provide tight bounds on the size of the relaxed skyline of the cells. It is easy to see that a trivial bound is $|S_r(J)| \leq |J| \leq M^d$. We provide below a better upper bound, which is also tight.

Proposition 3.12. *Let $T \subseteq G_d = [M_1] \times [M_2] \times \cdots \times [M_d]$. Then,*

$$|S_r(T)| \leq \prod_{i=1}^{d} M_i - \prod_{i=1}^{d} (M_i - 1)$$

Moreover, the bound is tight.

Proof. We start by proving that the bound is tight. For that, consider the set $J = T_0$, where T_0 is the following set in the case of d dimensions

$$T_0 = \{\mathbf{i} \in G_d \mid i_1 = 1 \vee i_2 = 1 \vee \dots i_d = 1, 1 \leq i_j \leq M_j\}$$

Intuitively, T_0 includes the d hyperplanes that pass through the point $(1, 1, \dots, 1)$. We will first count the size of T_0. Notice that T_0 contains all points of the grid that contain a coordinate with value 1. It is easy to count the points that do not contain any 1, since these points have $M_j - 1$ choices of values for the j-th coordinate. Hence, there are $\prod_{i=1}^{d}(M_i - 1)$ of these points. Since the total number of points is $\prod_{i=1}^{d} M_i$, it follows that

$$|T_0| = \prod_{i=1}^{d} M_i - \prod_{i=1}^{d}(M_i - 1)$$

Next, consider a point $\mathbf{i} \in T_0$. By construction, there exists some k such that $i_k = 1$. For any other point $\mathbf{i}' \in T_0$, $i_k = 1 \leq i'_k$; hence, $\mathbf{i}$ is not strictly dominated by any other point and $\mathbf{i} \in S_r(T_0)$. This implies that $T_0 \subseteq S_r(T_0)$. Since also $S_r(T_0) \subseteq T_0$, it follows that $S_r(T_0) = T_0$. Thus, $|S_r(T_0)| = |T_0|$ and the bound is indeed tight.

Next, we prove the upper bound. Consider an arbitrary $T \subseteq G_d$. We will show that there is an injective mapping from $S_r(T)$ to T_0, which proves that $|S_r(T)| \leq |T_0|$. Indeed, consider the following mapping:

$$map(i_1, \dots, i_d) = (i_1 - m, \dots, i_d - m)$$
$$\text{where} \quad m = \min\{i_1, \dots, i_d\} - 1$$

Intuitively, each point is mapped along the diagonal to the point where the diagonal intersects with T_0. First, we have to show that for every $\mathbf{i} \in S_r(T)$, $map(\mathbf{i}) \in T_0$. Indeed, for any $j = 1, d$, we have that $(map(\mathbf{i}))_j = i_j - \min\{i_1, \dots, i_d\} + 1 \geq 1$ and also $(map(\mathbf{i}))_j \leq i_j \leq M_j$. Moreover, the coordinate that is minimum will be equal to one.

Next, we have to show that the mapping is injective. Consider two points $\mathbf{i} \neq \mathbf{i}' \in S_r(T)$ such that $map(\mathbf{i}) = map(\mathbf{i}') = (v_1, \dots, v_d) = \mathbf{v}$. Then, by construction $\mathbf{i} = \mathbf{v} + m$ and $\mathbf{i}' = \mathbf{v} + m'$, for some m, m'. Without loss of generality assume that $m > m'$. Then, for every coordinate k, we have that $i'_k = v_i + m' < v_i + m = i_k$, hence $\mathbf{i}' \prec \mathbf{i}$ and $\mathbf{i} \notin S_r(T)$, which is a contradiction. □

In our setting, we have that $M_1 = \dots = M_d = M$, hence:

COROLLARY 3.13. *If $J \subseteq [M]^d$, then*

$$|S_r(J)| \leq M^d - (M-1)^d = O(M^{d-1})$$

4. ALGORITHMS

In this section, we use the tools we have developed in Section 3 to design three algorithms for parallel skyline query processing.

4.1 A 2-Step Algorithm with No Replication

Here, we propose a simple algorithm that operates in two steps. We choose $M = P$; hence, the total amount of data communicated at the broadcast phase is $dP(P+1) + P^{d+1}$, independent of n. The algorithm is based on Lemma 3.10, and, more precisely, on the fact that each cell needs to access data only from cells with at least one shared coordinate. Before we describe the algorithm in full detail, we need some definitions. For $k = 1, \dots, d$ and $s = 1, \dots, P$, let

$$R_{k,s} = \{\mathbf{x} \in R \mid \mathbf{x} \in B(\mathbf{i}), \mathbf{i} \in S_r(J), i_k = s\}$$

Intuitively, $R_{k,s}$ includes all the points that belong to cells of the relaxed skyline that are on the hyperplane $X_k = s$. The following lemma is straightforward.

LEMMA 4.1. *For any $k = 1, \dots, d$ and $s = 1, \dots, P$, we have $|R_{k,s}| \leq |B_k^s|$.*

We next define for any dimension $k = 1, \dots, d$

$$S^k(R) = \bigcup_{s=1}^{P} S(R_{k,s})$$

The following lemma captures the connection between the sets $S^k(R)$ and the skyline $S(R)$.

LEMMA 4.2. $S(R) = \bigcap_{i=1}^{d} S^i(R)$

PROOF. Let $\mathbf{x} \notin S(R)$ and $\mathbf{x} \in B(\mathbf{i})$. If $\mathbf{i} \notin S_r(J)$, then $\mathbf{x} \notin R_{k,s}$ for any k, s; hence $\mathbf{x} \notin \bigcap_{i=1}^{d} S^i(R)$. Otherwise, it must be that $\mathbf{i} \in S_r(J)$. Since $\mathbf{x}$ does not belong in the skyline of R, there exists some point $\mathbf{x}' \in S(R)$ such that $\mathbf{x}' \preceq \mathbf{x}$. Let $\mathbf{x}' \in B(\mathbf{j})$. By Lemma 3.9, $\mathbf{j} \in S_r(J)$. By applying Lemma 3.10, we also obtain that $\mathbf{j}, \mathbf{i}$ must have at least one coordinate in common, let it be the k-th. It follows that $\mathbf{x}, \mathbf{x}' \in R_{k,i_k}$. But then $\mathbf{x} \notin S(R_{k,i_k})$, since it is dominated by $\mathbf{x}'$. Moreover, $\mathbf{x} \notin R_{k,s}$ for $s \neq i_k$. Hence, $\mathbf{x} \notin S^k(R)$ and $\mathbf{x} \notin \bigcap_{i=1}^{d} S^i(R)$. It follows that $\bigcap_{i=1}^{d} S^i(R) \subseteq S(R)$.

Next, let $\mathbf{x} \in S(R)$ and $\mathbf{x} \in B(\mathbf{i})$. By Lemma 3.9, $\mathbf{i} \in S_r(J)$. For a dimension k, $\mathbf{x} \in R_{k,i_k}$ and, since there exists no point that dominates $\mathbf{x}$, it must hold that $\mathbf{x} \in S(R_{k,i_k})$. Hence, $\mathbf{x} \in S^k(R)$ for any $k = 1, \dots, d$. It follows that $\mathbf{x} \in \bigcap_{i=1}^{d} S^i(R)$ and $S(R) \subseteq \bigcap_{i=1}^{d} S^i(R)$. □

Lemma 4.2 gives a straightforward 2-step algorithm for the skyline computation. First, observe that, for a fixed k, $S^k(R)$ can be computed in one communication step, since we can choose $M = P$ and then assign the computation of $S(R_{k,s})$ to server s. Since the size of $R_{k,s}$ is $O(n/P)$, the first step is load-balanced. Now, notice that we can perform this computation in parallel for all the dimensions in the first step. The second step will compute the intersection of the sets $S^k(R)$. The detailed algorithm is as follows.

```
Algorithm 1: 2-STEP ALGORITHM
  STEP 1
  Broadcast: Compute the R-skyline (see Section 3)
  Communication: Server $s$ receives $R_{1,s}, R_{2,s}, \dots, R_{d,s}$
  Computation: Server $s$ computes $S(R_{1,s}), \dots, S(R_{d,s})$
  STEP 2
  Compute the set intersection [13]: $\bigcap_{s=1}^{d} S^s(R)$
```

THEOREM 4.3. *The* 2-STEP ALGORITHM *computes $S(R)$ in two steps and is perfectly load-balanced.*

PROOF. We first prove the correctness of the algorithm. Notice that we have not specified directly how $S^k(R)$ is computed. Instead, $S^k(R)$ is computed implicitly, since the sets $S(R_{k,s})$ for $s = 1, \dots, P$ are disjoint. Hence, by the end of step 1, $S^k(R)$ is partitioned among the servers. The correctness of the algorithm then follows directly from Lemma 4.2. It remains to prove that the algorithm is load-balanced.

Indeed, from Lemma 4.1, we obtain that $|R_{k,s}| = O(n/P)$ for any k, s. It follows that any server s receives total data of size $d \cdot O(n/P)$. Finally, the intersection of multiple sets can be computed in 1 step by a load-balanced algorithm, as proved in [13]. We should also note that the set intersection requires a randomized algorithm that uses a hash function to distribute the tuples among the servers. □

The 2-step algorithm replicates the data d times during the first step. There exists a simple variation of the 2-step algorithm which reduces the replication per step to a constant factor, independent of the dimension d, but with the tradeoff of having to increase the number of communication steps. Indeed, instead of computing the sets $S^i(R)$ in parallel, we can compute only $S^1(R)$ at the first step, $S^2(S^1(R))$ at the second step, and so on. The correctness of this algorithm follows from Lemma 4.2. Moreover, it is easy to see that the computation now requires d parallel steps.

4.2 A 1-step Algorithm with $O(P^{\frac{d-2}{d-1}})$ Replication

In this section, we describe a one step algorithm that achieves a load per server of $O(n/P \cdot P^{\frac{d-2}{d-1}})$. In other words, the data is replicated on average by a factor of at most $P^{\frac{d-2}{d-1}}$.

We first choose the number of partition points to be $M = P^{1/d-1}$. Thus, the total amount of data communicated during the broadcast phase is $dP(P+1) + P^{1+d/(d-1)}$, independent of n. By applying Corollary 3.13, it follows that the total number of cells in $S_r(J)$ will be at most $O((P^{1/d-1})^{d-1}) = O(P)$. Hence, we can assign to each server a constant number of cells. Let C_s be the set of cells assigned to server s. Each server is responsible for outputting only the points of the cells in C_s that belong to the final skyline.

However, we have to make sure that each server receives not only the data in C_s, but also data from other cells. It follows from Lemma 3.10 that the data in the cells of the set $N(\mathbf{i})$ is sufficient to compute $S(R) \cap B(\mathbf{i})$.

```
Algorithm 2: 1-Step Replication
  Broadcast: Compute the R-skyline (see Section 3)
  Communication: Server $s$ receives $C_s$. Moreover, for
  every $\mathbf{i} \in C_s$, it receives $B(\mathbf{j})$ for every $\mathbf{j} \in N(\mathbf{i})$.
  Computation: Server $s$ computes
  $S(R) \cap \{\mathbf{x} \in B(\mathbf{i}) \mid \mathbf{i} \in C_s\}$
```

We next prove the correctness of the algorithm and compute the load per server.

Theorem 4.4. *The* 1-Step Replication *algorithm computes $S(R)$ in one step with a maximum load guarantee of $O(n/P^{1/(d-1)})$.*

Proof. We have already shown that each server holds sufficient information to decide whether a point in $\{\mathbf{x} \in B(\mathbf{i}) \mid \mathbf{i} \in C_s\}$ belongs in the final skyline.

In order to compute the load per server, let us consider a server s and let $\mathbf{i} \in C_s$. We will compute an upper bound on the size of $D(\mathbf{i}) = \bigcup_{\mathbf{j} \in N(\mathbf{i})} B(\mathbf{j})$. Indeed, we can partition $N(\mathbf{i})$ in the sets $\bigcup_{\mathbf{j} \in N(\mathbf{i}), j_k = i_k} B(\mathbf{j})$ for $k = 1, \ldots, d$. Now, since each such set contains cells with a common coordinate, we have that $|\bigcup_{\mathbf{j} \in N(\mathbf{i}), j_k = i_k} B(\mathbf{j})| \le |B_k^{i_k}| = O(n/P^{1/d-1})$.

It follows that $|D(\mathbf{i})| = d \cdot O(n/P^{1/d-1})$. Moreover, for every server s, $|C_s|$ is bounded by some constant. Hence, the load for each server is bounded by $O(n/P^{1/d-1}) = O(n/P \cdot P^{\frac{d-2}{d-1}})$. $\Box$

Corollary 4.5. *The* 1-Step Replication *algorithm is perfectly load-balanced in 2 dimensions.*

Even though the algorithm is load-balanced for 2 dimensions, for any $d > 2$ the load per server is much higher. For example, for $d = 3$, the replication is on average $O(\sqrt{P})$. In the next section, we propose a specialized algorithm that keeps the load per server to $O(n/P)$ for $d = 3$.

4.3 A 1-Step Algorithm with No Replication for 3D

In this section, we propose and analyze a load-balanced algorithm for 3-dimensional data sets. We present two variants of the algorithm: the first variant uses randomization and requires $M = P \log P$, whereas the second variant is deterministic and requires $M = P$. In both cases, the amount of communication during the broadcast phase is constant, independent of n.

Let the database be $R(X, Y, Z)$. The algorithm exploits the following observation.

Lemma 4.6. *Let $\mathbf{j}, \mathbf{i} \in S_r(J)$ such that $\mathbf{j} \preceq \mathbf{i}$. Also, suppose that $\mathbf{i}$ and $\mathbf{j}$ share exactly one coordinate (let it be the k-th). Let $\mathbf{x} \in B(\mathbf{i})$. Then, $\mathbf{x}$ is not dominated by some point in $B(\mathbf{j})$ if and only if $\mathbf{x}_k < m_k(\mathbf{j}) = \min_{\mathbf{x}' \in B(\mathbf{j})} \mathbf{x}'_k$.*

Proof. Consider the point $\mathbf{x}' \in B(\mathbf{j})$ such that $\mathbf{x}' = \arg\min_{\mathbf{x}' \in B(\mathbf{j})} \mathbf{x}'_k$. For any other coordinate $\ell \neq k$, since $j_k < i_k$, it is clear that $\mathbf{x}'_\ell < \mathbf{x}_\ell$ (it also holds that $\mathbf{x}_\ell \neq \mathbf{x}'_\ell$). If $\mathbf{x}_k \ge m_k(\mathbf{j})$, we also have $\mathbf{x}_k \ge \mathbf{x}'_k$ and thus $\mathbf{x}' \preceq \mathbf{x}$.

For the other direction, if $\mathbf{x}_k < \min_{\mathbf{x}' \in B(\mathbf{j})} \mathbf{x}'_k$, then $\mathbf{x}$ strictly dominates every point in $B(\mathbf{j})$ along the k-th coordinate; hence, it can not be dominated by any point in $B(\mathbf{j})$. $\Box$

Now, let us consider a cell $\mathbf{i} \in S_r(J)$. For any $\mathbf{j} \in N(\mathbf{i})$ such that $\mathbf{i}, \mathbf{j}$ coincide in exactly one coordinate (the k-th), Lemma 4.6 implies that $\mathbf{i}$ needs to hear only the value $m_k(\mathbf{j})$ from the cell $B(\mathbf{j})$. The algorithm computes for each cell $\mathbf{i} \in S_r(J)$ the values $m_x(\mathbf{i}), m_y(\mathbf{i}), m_z(\mathbf{i})$ and then broadcasts this data. Since Corollary 3.13 implies that the number of cells in the relaxed skyline for 3 dimensions is $O(P^2)$, it follows that each server will have an extra constant load of $3 \cdot O(P^2)$.

The computation of the minimum coordinates for each cell can be integrated in the broadcast phase. After bucketizing, each server s sends, for every cell $\mathbf{i} \in S_r(J)$, the local minimum value of each coordinate, which we denote by $m_k^s(\mathbf{i})$ for the k-th dimension. The total data communicated is $3 \cdot |S_r(J)| \cdot P = O(P^3)$. The minimum value for cell $B(\mathbf{i})$ is then computed as $m_k(\mathbf{i}) = \min_s m_k^s(\mathbf{i})$.

We say that two cells $\mathbf{i}, \mathbf{j}$ are *colinear*, denoted by $\mathbf{i} \wr \mathbf{j}$, if they share exactly two coordinates. Then,

Corollary 4.7. *If each server has available the values $m_x(\mathbf{j}), m_y(\mathbf{j}), m_z(\mathbf{j})$ for each cell $\mathbf{j} \in S_r(J)$, a cell $\mathbf{i} \in S_r(J)$ needs to access data only from cells in*

$$N^r(\mathbf{i}) = \{\mathbf{j} \in S_r(J) \mid \mathbf{j} \preceq \mathbf{i},\ \mathbf{i} \wr \mathbf{j}\}$$

Notice that $N^r(\mathbf{i}) \subseteq N(\mathbf{i})$. Hence, the data each cell needs in order to compute the actual skyline from the local skyline is substantially reduced.

We need to note here that, in the case of $d > 3$ dimensions, even after the minimum values for each dimension are computed and sent to every server, it is not sufficient for a cell to access only colinear cells to compute the skyline.

For this reason, our algorithm does not carry over to larger dimensions.

The next step of the algorithm is to compute a partition $\mathcal{G}$ of the set $S_r(J)$, such that each set $G_\ell \in \mathcal{G}$, which we call a *group*, is responsible for computing the skyline points only in $S(R) \cap \bigcup_{\mathbf{i} \in G_\ell} B(\mathbf{i})$. In order to perform this computation, each group G_ℓ must also obtain the set of points

$$D(G_\ell) = \bigcup_{\mathbf{i} \in G_\ell} (\cup_{\mathbf{j} \in N^r(\mathbf{i})} B(\mathbf{j}))$$

We will show how to construct a partition $\mathcal{G}$ which satisfies the following properties:

1. Each group $G_\ell \in \mathcal{G}$ has a limited amount of data: more precisely, $|D(G_\ell)| = O(n/M)$

2. The total amount of data in the groups is
$$\sum_{G_\ell \in \mathcal{G}} |D(G_\ell)| = O(n)$$

Given such a partition $\mathcal{G}$, we can allocate the groups to servers such that the computation is load-balanced. We postpone the discussion on how to perform the assignment of groups to servers for the end of this subsection.

We now describe the construction of the desired partition. The algorithm distinguishes two disjoint classes of cells in the relaxed skyline: *interior cells* and *corner cells*. We treat each class of cells in a distinct way; more precisely, the partition $\mathcal{G}$ can be defined as $\mathcal{G} = \mathcal{G}^{in} \cup \mathcal{G}^{co}$, where $\mathcal{G}^{in}, \mathcal{G}^{co}$ are the partitions of the interior and corner cells respectively.

DEFINITION 4.8. *A cell* $\mathbf{i} \in S_r(J)$ *is an* interior cell *if every cell in* $N^r(\mathbf{i})$ *belongs to the same plane. We also say that* $\mathbf{i}$ *is interior to this specific plane. A* corner cell *is a cell that is not interior.*

Figure 2 shows the relaxed skyline of a 3-dimensional data set, along with the interior and corner cells. The R-skyline consists only of the visible cells. Although one may be tempted to think that cells may be interior to at most one plane, this is not always true. For example, consider the relaxed skyline consisting only of the cells $\mathbf{i} = (1,1,1)$ and $\mathbf{j} = (1,1,2)$. Clearly, $\mathbf{i} \in N^r(\mathbf{j})$. It follows that $\mathbf{j}$ is an interior cell for both the planes $X = 1$ and $Y = 1$.

The interior cells can be easily handled by our algorithm. Indeed, for $i = 1, \ldots, M$, we can assign to a distinct group G_i^{in} the cells interior to the planes $X = i$, $Y = i$ and $Z = i$. If a cell $\mathbf{i}$ is interior to more than one plane, we assign it to the group $G_{\mathbf{i}_x}^{in}$ if it is interior to the plane $X = \mathbf{i}_x$; else, we assign it to the group $G_{\mathbf{i}_y}^{in}$. By definition, the interior cells need to be informed about data only from the plane they are interior to. Thus, it suffices to send every cell of the planes $X = i$, $Y = i$ and $Z = i$ to the group G_i^{in}. Since each plane holds at most $O(n/M)$ data, each group will hold $3 \cdot O(n/M)$ data. Furthermore, the total size of the data in these groups will be $M \cdot O(n/M) = O(n)$.

Next, we show how to process the corner cells. We will treat the corner cells by grouping them into *lines*.

DEFINITION 4.9. *A* line $L(\ell_x, \ell_y)$, *where* $1 \leq \ell_x, \ell_y \leq M$ *is the set of corner cells* $\mathbf{i}$ *such that* $\mathbf{i}_x = \ell_x$, $\mathbf{i}_y = \ell_y$ *and* $|L(\ell_x, \ell_y)| > 1$. *We similarly define* $L(\ell_x, \ell_z)$ *and* $L(\ell_y, \ell_z)$.

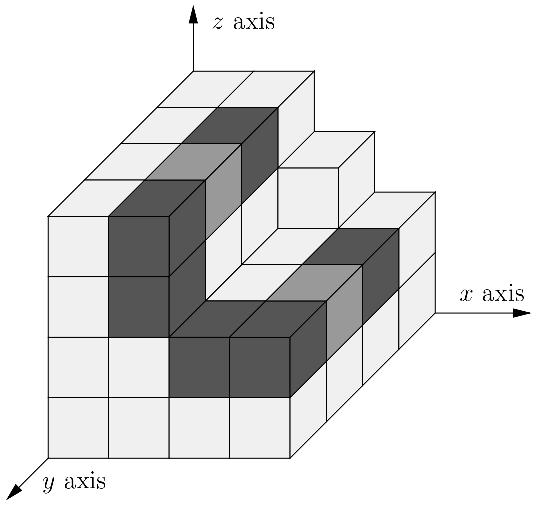

Figure 2: The R-skyline of a 3-dimensional data set. The dark grey cells are border cells, the lighter grey cells are just corner cells and the rest are interior cells.

In our example in Figure 2, the corner cells form 4 lines. Notice that a corner cell may belong to more than one line (at most one at each direction). It is easy to see that a corner cell either belongs to at least one line or it is not colinear with any other corner cell. We call the latter a *single* corner cell. Before we give some useful properties of the lines, let us present the following lemma.

LEMMA 4.10. *If* $\mathbf{i}$ *is a corner cell, then for every dimension* k, *there exists a cell* $\mathbf{j} \in N^r(\mathbf{i})$ *such that* $\mathbf{j}_k < \mathbf{i}_k$.

PROOF. Fix a dimension k and assume that for every $\mathbf{i}' \in N^r(\mathbf{i})$, $\mathbf{i}'_k = \mathbf{i}_k$. Then, all cells of $N^r(\mathbf{i})$ would belong in the same plane, a contradiction since $\mathbf{i}$ is not an interior cell. □

We are particularly interested in corner cells with specific properties.

DEFINITION 4.11. *A corner cell is a* border *cell if it is maximal or minimal for every line that it belongs to.*

In Figure 2, the border cells are the cells colored in dark grey. An easy observation about border cells is that each line has exactly two border cells. The key property about border cells is that an intersection of two lines is always a border cell. This property heavily depends on the fact that the cells form an R-skyline and does not hold for any collection of cells.

LEMMA 4.12. *If a corner cell belongs to two distinct lines, then it is a border cell.*

PROOF. Let $\mathbf{i} \in S_r(J)$ be a corner cell. Without loss of generality, let us assume that it belongs to the lines $L_1(\mathbf{i}_x, \mathbf{i}_y)$ and $L_2(\mathbf{i}_x, \mathbf{i}_z)$. Now, for the sake of contradiction, suppose that $\mathbf{i}$ is not a border cell; then, for some line, let it be L_1, there exist cells $\mathbf{i}', \mathbf{i}''$ such that $\mathbf{i}'_z < \mathbf{i}_z$ and $\mathbf{i}''_z > \mathbf{i}_z$.

Now, consider a corner cell $\mathbf{j} \neq \mathbf{i}$ in line L_2. It is easy to see that $\mathbf{j}_x = \mathbf{i}_x$ and $\mathbf{j}_z = \mathbf{i}_z$. We distinguish two cases: $\mathbf{j}_y > \mathbf{i}_y$ and $\mathbf{j}_y < \mathbf{i}_y$ (note that $\mathbf{i}_y \neq \mathbf{j}_y$).

For the first case, by applying Lemma 4.10, there exists a cell $\mathbf{j}' \in N^r(\mathbf{i}')$ such that $\mathbf{j}'_x < \mathbf{i}_x$. Now, we have that: $\mathbf{j}'_x < \mathbf{i}'_x = \mathbf{i}_x = \mathbf{j}_x$, $\mathbf{j}'_y = \mathbf{i}'_y = \mathbf{i}_y < \mathbf{j}_y$ and $\mathbf{j}'_z = \mathbf{i}'_z < \mathbf{i}_z = \mathbf{j}_z$. Thus, $\mathbf{j}' \prec \mathbf{j}$, which is a contradiction, since $\mathbf{j} \in S_r(J)$.

For the second case, by applying again Lemma 4.10, there exists a cell $\mathbf{j}' \in N^r(\mathbf{j})$ such that $\mathbf{j}'_x < \mathbf{j}_x$. However, $\mathbf{j}'_x < \mathbf{j}_x = \mathbf{i}_x = \mathbf{i}''_x$, $\mathbf{j}'_y = \mathbf{j}_y < \mathbf{i}_y = \mathbf{i}''_y$ and $\mathbf{j}'_z = \mathbf{j}_z = \mathbf{i}_z < \mathbf{i}''_z$. Thus, $\mathbf{j}' \prec \mathbf{i}''$, a contradiction. □

We can now discuss how the corner cells are partitioned into groups. We assign each single corner cell, i.e. a corner cell that does not belongs to any line, and each line to a distinct group. If a cell belongs to more than one line, we assign the cell to the group of the lexicographically first line. We next prove the validity of this partitioning.

Lemma 4.13. *For each group $G^{co}_\ell \in \mathcal{G}^{co}$, we have that $|D(G^{co}_\ell)| = O(n/M)$.*

Proof. If the group G^{co}_ℓ consists of a single cell, then the cell needs to receive data only from three lines, and each line holds at most $O(n/M)$ data by construction. In the case that the group is a line, let it be $L(\ell_x, \ell_y)$, the cells that belong in $\bigcup_{\mathbf{i}\in L} N^r(\mathbf{i})$ reside either on the plane $X = \ell_x$ or on the plane $Y = \ell_y$. Since each plane contains $O(n/M)$ data, it follows that G^{co}_ℓ must hold at most $O(n/M)$ data. □

Lemma 4.14. *$\sum_{G^{co}_\ell \in \mathcal{G}^{co}} |D(G^{co}_\ell)| = O(n)$.*

Proof. Consider a cell $\mathbf{i} \in S_r(J)$. By our construction, $\mathbf{i}$ must send its data to any corner cell $\mathbf{j} \in S_r(J)$ that differs with $\mathbf{i}$ only in one dimension. Let us fix the dimension to be X and let $T_x(\mathbf{i})$ be the set of cells where $\mathbf{i}$ needs to send its data.

The first case is that there exists only one corner cell $\mathbf{j} \in T_x(\mathbf{i})$. Since $\mathbf{j}$ belongs to exactly one group, the data $B(\mathbf{i})$ will be sent only to one group along the X dimension.

Otherwise, $T_x(\mathbf{i})$ has at least two cells. Then, the cells in $T_x(\mathbf{i})$ define a line $L_1 = L(\mathbf{i}_y, \mathbf{i}_z)$. However, it is not necessary that all the cells of L_1 are assigned to the group which represents L_1. Thus, we need to bound the number of groups n_ℓ that include cells from L_1. More precisely, we show that $n_\ell \leq 3$.

Indeed, notice that any other line that includes a cell from L_1 intersects L_1. However, Lemma 4.12 tells us that lines can intersect only on border cells. Clearly, a line has at most 2 border cells; hence, at most 2 cells may belong to other groups.

Hence, the replication of a cell across dimension X is at most 3. Summing up for all 3 dimensions X, Y, Z, we conclude that the replication of any cell is at most 9. Since the data in each cell is replicated a constant number of times, the total data sent will be $O(n)$. □

We now return to the task of allocating the groups of the parition $\mathcal{G}$ to the servers. We propose two algorithms for this task: a randomized algorithm (R-Allocate) and a deterministic algorithm (D-Allocate).

We prove the load balancing of algorithm R-Allocate by using tools from the balls-into-bins framework. Indeed, one can view each group $G_\ell \in \mathcal{G}$ as a weighted ball, where its weight is $|D(G_\ell)|$, and each server as a bin. Then, the algorithm chooses for each ball a bin independently and uniformly at random (u.a.r.) and places the ball into this bin.

Procedure R-Allocate($\mathcal{G}$)

```
• $M \leftarrow P \log P$
• Assign each group independently to a uniformly at
  random chosen server.
```

Proposition 4.15. *Assume P bins and weighted balls of total size $O(n)$ such that the maximum weight of a ball is $w_{max} = O(n/P \log P)$. If the balls are thrown independently and u.a.r. in the bins, each bin holds a total weight of $O(n/P)$ with high probability (w.h.p.).*

Proof. By applying the majorization lemma in [2], it follows that the worst balancing occurs in the case we have $N = n/w_{max}$ balls, each one with weight w_{max}. If $N \leq P \log P$, then the maximum number of balls landing on any bin will be w.h.p. at most $O(\log P)$. Hence, the maximum weight will be bounded w.h.p. by $O(\log P) \cdot w_{max} = O(n/P)$. In the case where $N \geq P \log P$, applying the theorem in [19], the maximum number of balls will be w.h.p. $O(N/P)$. It thus follows that the maximum total weight at any server will be w.h.p. $O(N/P) \cdot w_{max} = O(n/P)$. □

The deterministic algorithm D-Allocate needs first to count the amount of data at each cell. This task can be integrated in the broadcast phase: each server, apart from the minimum values, also reports the number of points in each cell.

The algorithm chooses $M = P$. Let us assume that the partition guarantees that for each $G_\ell \in \mathcal{G}$, $|D(G_\ell)| \leq c_1 n/P$ and also $\sum_{G_\ell \in \mathcal{G}} |D(G_\ell)| \leq c_2 n$.

Procedure D-Allocate($\mathcal{G}$)

```
• $M \leftarrow P$
• $c = \max\{c_1, c_2\}$
• For each $G_\ell \in \mathcal{G}$, assign group $G_\ell$ to the first server
  with data less or equal to $cn/P$
```

Proposition 4.16. D-Allocate *distributes the groups such that each server receives $O(n/P)$ data.*

Proof. We will show that: (1) every group is assigned to some server, and (2) each server receives at most $2cn/P$ data.

For the first part, suppose that some group $G_\ell \in \mathcal{G}$ can not be assigned to any server. It follows that every server has strictly more than cn/P data. Hence, the total data in the servers will be $> P(cn/P) = cn \geq c_2 n$, a contradiction.

For the second part, consider a server with data strictly more than $c \cdot n/P$ and consider the last group assigned to it. Before this assignment, the server had received at most cn/P data. However, the maximum size of a group is $c_1 n/P \leq cn/P$. Hence, the server holds at most $2cn/P$ data. □

We summarize the algorithm for the 3-dimensional case in Algorithm 5. The algorithm and the analysis also imply the main theorem for this subsection.

Theorem 4.17. *There exists a perfectly load-balanced algorithm that computes the skyline $S(R)$ for 3 dimensions in one step.*

Algorithm 3: 1-Step Algorithm

Broadcast:

- Compute the R-skyline (see Section 3)
- Compute the minimum values m_x, m_y, m_z for each cell
- Compute the balanced partition $\mathcal{G}$

Communication:

- Broadcast the minimum values
- Apply R-Allocate($\mathcal{G}$) or D-Allocate($\mathcal{G}$)

Computation:

- For each G_ℓ in server s, compute $S(R) \cap \bigcup_{\mathbf{i} \in G_\ell} B(\mathbf{i})$

5. CONCLUSION

In this paper we presented three algorithms for computing the skyline on parallel server clusters. Our algorithms need only one or two synchronization steps, and are provably load-balanced. We leave open the question whether the skyline can be computed in one single synchronization steps, with perfect load balancing (we could only solve this problem for $d \leq 3$ dimensions).

Our work is part of a broader effort to design efficient algorithms for data processing on parallel server clusters, along the lines of [1, 13]. While that work has studied only Conjunctive Queries, the skyline operator represents a case that extends Conjunctive Queries with both order predicates and one level of negation: for example, in two dimensions, the skyline query can be expressed as

$$S(x, y) = R(x, y), \neg\exists u, v.(R(u, v), u \leq x, v \leq y)$$

In future work, we plan to study the computation of other classes of queries on parallel server clusters.

6. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010.

[2] P. Berenbrink, T. Friedetzky, Z. Hu, and R. A. Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, 2008.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430. IEEE Computer Society, 2001.

[4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816. IEEE Computer Society, 2003.

[5] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh. Parallel computation of skyline queries. In *HPCS*, page 12. IEEE Computer Society, 2007.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] F. K. H. A. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Symposium on Computational Geometry*, pages 298–307, 1993.

[8] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[9] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240. ACM, 2005.

[10] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[11] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948. SIAM, 2010.

[12] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD Conference*, pages 85–96. ACM, 2011.

[13] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234. ACM, 2011.

[14] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[15] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in z order. In *VLDB*, pages 279–290. ACM, 2007.

[16] J. Matousek. Computing dominances in E^n. *Inf. Process. Lett.*, 38(5):277–278, 1991.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[18] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *ICDE*, pages 760–771. IEEE, 2009.

[19] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *RANDOM*, pages 159–170, 1998.

[20] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Agids: A grid-based strategy for distributed skyline query processing. In *Globe*, volume 5697 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 2009.

[21] I. Stojmenovic and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2):249–251, 1988.

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

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

[24] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2006.