# Computing the Skyline of a Relational Table Based on a Query Lattice

Nicolas Spyratos, Tsuyoshi Sugibuchi, Ekaterina Simonenko, and Carlo Meghini

Laboratoire de Recherche en Informatique, Université Paris-Sud 11, France
{Nicolas.Spyratos, Tsuyoshi.Sugibuchi, Ekaterina.Simonenko}@lri.fr
Istituto di Scienza e Tecnologie della Informazione del CNR, Pisa, Italy
Carlo.Meghini@isti.cnr.it

**Abstract.** We propose a novel approach to computing the skyline set of a relational table $R$, with respect to preferences expressed over one or more numerical attributes. Our approach is based on what we call the *query lattice* of $R$, and our basic algorithm constructs the skyline set as the union of the answers to a subset of queries from that lattice - hence *without* directly accessing the table R. Therefore, in contrast to all existing techniques, our approach is independent of how the table R is implemented or how its tuples are indexed. We demonstrate the generality of our approach by computing the skyline set of the join of two tables based on the product of their individual query lattices - therefore *without* performing the join. The paper presents basic concepts and algorithms leaving experimentation and performance evaluation to a forthcoming paper.

**Keywords:** skyline, relational table, query lattice

## 1 Introduction

In many multicriteria decision-making applications, dominance analysis is an important aspect. As an example, consider a person looking for a vacation package using two criteria, or "attributes": hotel rating and price. Intuitively, a package $P = \langle r, p \rangle$ is better than a package $P' = \langle r', p' \rangle$ if $P$ is better than $P'$ in one attribute and not worse than $P'$ in the other attribute. If this is the case then we say that $P$ *dominates* $P'$.

For example, consider the following three packages:

– $P_1 = \langle 2, 100 \rangle, P_2 = \langle 3, 130 \rangle, P_3 = \langle 2, 120 \rangle$

Since a higher rating and a lower price are more preferable, $P_1$ dominates $P_3$. On the other hand, $P_1$ and $P_2$ don't dominate each other because $P_1$ has a lower rating than $P_2$ and $P_2$ has a higher price. Similarly, $P_2$ and $P_3$ don't dominate each other because $P_2$ has a higher rating than $P_3$ and $P_3$ has a lower price.

A package, or "tuple" that is not dominated by any other tuple is said to be a skyline tuple or to be in the skyline. The tuples in the skyline are the best possible trade-offs among the attribute values appearing in the tuples. Thus in our example, packages $P_1$ and $P_2$ are in the skyline, while $P_3$ is not.

In order to conduct a skyline analysis, two items must be specified:

1. A set of attributes over which preferences are expressed (such as Hotel Rating and Price, in our example).
2. An ordering of the attribute values (either total or partial) according to which preference is expressed (such as the ordering of the integers for Hotel Rating to express that rating $r$ is preferred to rating $r'$ if $r > r'$; and similarly for Price to express that price $p$ is preferred to price $p'$ if $p < p'$).

In recent years, skyline analysis has gained considerable interest in the area of information systems in general, and in the area of databases in particular. However, skyline analysis (i.e. computing non dominated points) existed well before the concept appeared in database research; it is known as the maximum vector problem or the Pareto optimum [11][18]. The popularity of skyline analysis in the area of information systems is mainly due to its applicability for decision making applications. Indeed, as information systems store larger and larger volumes of data today, data management and in particular query processing present difficult challenges. From the user viewpoint, large volumes of data imply answers of large size. By returning the best tuples (in terms of user preferences), the skyline query relieves the user from having to deal with answers of large size in order to find the best tuples.

The skyline operator was first introduced in [3], where the authors also present two basic algorithms: the Block Nested Loops (BNL) and the Divide and Conquer (D&C). In order to improve the performance of BNL algorithm, the SFS (*sort-filter-skyline*) algorithm was proposed in [5]. SFS runs on data sorted according to a monotonic function (namely, *entropy* descending). Such sorting guarantees the non-dominance of each object by those that follow in the order. Therefore, once an object is put into the buffer window, it can be reported as part of the skyline. Not only this makes the SFS algorithm progressive, but also allows to reduce the number of comparisons needed, since SFS compares only against the non-dominated tuples, whereas BNL often compares against dominated tuples [7]. Despite this improvement, all objects have to be scanned by the algorithm at least once. Succeeding approaches tend to avoid scanning the complete data set. Namely, SaLSa [1] uses the *minimal coordinate* of each object as a sorting function, and during the filter-scan step checks if all remaining objects are dominated by a so-called *stop* object, which can be determined in $O(1)$ from the data accessed so far. The shortcoming is that the performance of SaLSa algorithm is affected by the data distribution and increasing dimensionality, since in higher dimensions instances of the problem the pruning power of the stop object is limited [29].
More generally, all sort-based techniques share the same drawback, namely the number of computations during the filter-scan step, as every input object should be compared with the skyline points in the buffer (which can potentially become large).

An alternative to the sort-based techniques is the use of indexes, which allows to avoid scanning all the input objects. The basic idea is to rely on an index in order to determine dominance between tuples, and to exclude tuples from further processing as early as possible. Two index-based algorithms, Bitmap and Index were first introduced in [21]. Bitmap uses the bitmap encoding of the data so that the dominating points are determined by a bit-wise "and" operation. The Index approach partitions the objects into a set of lists. Each list is sorted by minimum coordinate and indexed by a B-tree. The objects are accessed in batches defined by the values of the minimal coordinate, while the algorithm computes local skylines in each "batch" of the lists and then merges them into a global skyline. However, besides the computation cost of Bitmap, and the necessity to construct a B-tree for every combination of dimensions having potential interest for the user, the order in which skyline points are returned by these algorithms is fixed and depends on the data distribution, so it cannot be adapted to the users preferences.

Two other index-based skyline algorithms, NN (nearest-neighbor) [13] and BBS (*branch-and-bound skyline*) [19], are based on the observation that the object closest to the origin has to be part of the skyline. Nearest neighbor search is used to retrieve such point by using the R-tree. The pitfall of the NN algorithm is that in order to iteratively find the next nearest neighbors it divides the data set into overlapping partitions, and therefore duplicates have to be removed by traversing the R-tree multiple times. To avoid that, BBS rather accesses partially dominated nodes of the R-tree.

The main drawback of all index-based approaches is that not all data can be indexed (namely when data is dynamically produced). Also, R-trees and other multidimensional indexes have their own limitations, namely the curse of dimensionality.

Concerning related work specifically targeting the multi-relational skyline (or skyline join), two progressive algorithms are proposed in [9]. The idea is to combine the join with nested-loop and sort-merge algorithms. However, each relation has to accessed multiple times in order to compute the skyline for each join value, and then the global skyline. In addition, each input object has to be scanned at least once.

More recent work also considers how to compute the skyline over the join of two or more relational tables without actually computing the join [23]. Apart from its applicability to computing skylines in a centralized database, the interest of such work lies in the fact that it is also applicable to distributed environments. Earlier work related to this topic can be found in [1][2][8][10][20][25][26].

A lattice-based approach to single-relation skyline computation is introduced in [15], with the aim of proposing a data-distribution independent algorithm. A lattice structure is used to answer skyline queries over dimensions with low-cardinality domains, or those that can be mapped to low-cardinality domains (such as *Price*, that can be mapped to price ranges). The principle is to organize all the values combinations into a lattice based on the dominance relationship, and then to retrieve those that (a) are present in the input data set, and (b)

are not reachable by the dominance relationship from another element of the lattice, also belonging to the data set. However, no early pruning is done, so the entire data set has to be read twice in order to determine the skyline tuples, and the skyline join problem is not investigated. Also, the mapping of the values of a domain to a set of ranges has to be carefully tuned in order to deliver a meaningful skyline result, which is not discussed in the paper.
Several variants of skyline were introduced in [19], such as constrained, subspace and dynamic skyline queries (see also [6][16][22][27][28]). Skyline queries have also been studied in various other domains, outside traditional databases. These include probabilistic skyline computations over uncertain data [17](e.g. data in sensor networks); skyline computations over incomplete data [12](e.g. data with missing values); over data whose attributes have partially-ordered domains [4](e.g. preferences expressed by users online); over stream data[14]; or even bandwidth-constrained skyline computations over mobile devices [24].

In this paper, we present a novel approach to computing skylines which represents a major deviation from existing approaches. Indeed, instead of accessing individual tuples in a database table, our approach relies on the definition of skyline as the union of the answers to a set of queries. In doing so, our basic algorithm avoids accessing the table directly: access to the table is through queries, hence independent of how the table is implemented or how its tuples are indexed.

Given a relational table $R$, our approach is based on what we call the query lattice of $R$; and our basic algorithm constructs the skyline set as the union of the answers to a subset of queries from that lattice - hence *without* directly accessing the table $R$. We demonstrate the generality of our approach by computing the skyline of the join of two tables based on the product of their individual query lattices - therefore *without* performing the join. The paper presents basic concepts and algorithms leaving experimentation and performance evaluation to a forthcoming paper.

The paper is organized as follows. In section 2 we give some preliminary definitions and introduce our notation. In section 3 we present our basic algorithm for computing skylines through queries. In section 4 we apply our approach to computing skylines over joins, thus demonstrating the generality of the approach. Finally, in section 5, we offer some concluding remarks and discuss further research.

## 2 Basic definitions

Let $R$ be a relational table, with $A_1, \ldots, A_n$ as attributes. Let $\mathcal{B} = \{B_1, \ldots, B_k\}$, $k \leq n$, be the set of *preference attributes,* that is a set of attributes of the table whose domains are numeric and over which preferences are declared.

A *preference* over $B_i$ is an expression of one of two forms: $B_i \rightarrow min$ or $B_i \rightarrow max$. If the preference $B_i \rightarrow min$ is expressed by a user of the table, then this is interpreted as follows: given two values $x$ and $y$ in the domain of $B_i$, *$x$ is preferred to $y$* or *$x$ preceeds $y$* iff $x < y$; and similarly, if the preference $B_i \rightarrow max$

is expressed by a user, then this is interpreted as follows: given two values $x$ and $y$ in the domain of $B_i$, *$x$ is preferred to $y$* or *$x$ preceeds $y$* iff $x < y$;

In order to simplify the presentation, and without loss of generality, we shall consider only one form of preference, namely $B_i \rightarrow min$. However, all methods discussed in this paper can be applied with any combination of the preferences $B_i \rightarrow min$ and $B_i \rightarrow max$.Therefore, from now on, given two values $x$ and $y$ in the domain of $B_i$, we shall say that *$x$ is preferred to $y$* or *$x$ preceeds $y$* iff $x < y$.

**Definition 1** *(Pareto domination) Let $\mathcal{B} = \{B_1, \ldots, B_k\}$ be a set of preference attributes of a relational table $R$ and let $s$ and $t$ be tuples of $R$. We say that* s is equivalent to t, *denoted as $s \equiv t$ iff $s.B_i = t.B_i$ for all $B_i \in \mathcal{B}$. Moreover, we say that* s Pareto dominates t, *denoted as $s <_{Pa} t$, iff $s \not\equiv t$ and for all $B_i \in \mathcal{B}$, $s.B_i \leq t.B_i$.*

In order to simplify the presentation we will simply say “dominates” instead of “Pareto dominates”, and we shall drop the subscript in the notation, writing $s < t$ instead of $s <_{Pa} t$.

We shall call *Pareto preference query,* or simply *preference query* over $R$, any expression of the form $(B_1 = b_1) \wedge \ldots \wedge (B_k = b_k)$, where each $b_i$ is a value in the domain of attribute $B_i$. For simplicity of notation we shall denote a preference query simply by $\langle b_1, \ldots, b_k \rangle$.

Note that $\langle b_1, \ldots, b_k \rangle$ denotes also a tuple in the projection of $R$ over the preference attributes; however, context will always disambiguate. Also note that a preference query $\langle b_1, \ldots, b_k \rangle$ returns the set of tuples in $R$ whose projection over the preference attributes is the tuple $\langle b_1, \ldots, b_k \rangle$; therefore the answer to each query of the form $\langle b_1, \ldots, b_k \rangle$ is a Pareto equivalence class.

It is easy to verify that Pareto domination is irreflexive (*i.e.*, $s < s$ is false for each tuple $s$) and transitive (*i.e.*, $s < t$ and $t < u$ imply $s < u$ for all tuples $s$, $t$ and $u$), hence a strict order over $R$. A partial order $\leq$ over $R$ can be defined from Pareto domination as follows:

$$s \leq t \text{ iff } (s < t \text{ or } s = t)$$

for all tuples $s$ and $t$ in $R$. We shall say that *$s$ Pareto preceeds $t$*, or simply that *$s$ preceeds $t$*, whenever $s \leq t$.

Clearly, Pareto precedence defines a partial order also over preference queries. Moreover, given preference queries $s$ and $t$ we define the following operations:

– $s \otimes t = \langle min\{s.B_1, t.B_1\}, \ldots, min\{s.B_k, t.B_k\} \rangle$
– $s \oplus t = \langle max\{s.B_1, t.B_1\}, \ldots, max\{s.B_k, t.B_k\} \rangle$

It is easy to check that these operations make the set of preference queries over $R$ into a complete lattice, with $\otimes$ defining the least upper bound and $\oplus$ defining the greatest lower bound of any two preference queries. We notice that this lattice may be infinite, as some of the domains of the preference attributes may be infinite. However, if we require that each $b_i$ in a preference query be in the active domain of $B_i$ (*i.e.*if we require that each $b_i$ appear in $R$), then the lattice becomes finite and therefore it has a top and a bottom query, denoted as $\top$ and $\bot$, respectively. These extreme elements are given by:

$$\top = \langle m_1, \ldots, m_k \rangle$$
$$\bot = \langle M_1, \ldots, M_k \rangle$$

where $m_i$ and $M_i$ are the minimum value and the maximum value appearing in the active domain of $B_i$, respectively. We shall call this (finite) lattice the *query lattice,* of $R$ and we shall denote it as $(Q, \leq)$, where $Q$ is the (finite) set of preference queries over $R$.

In this paper, we shall use the query lattice for two purposes: (a) as a tool for computing skylines of relational tables, and (b) as a means for comparing our approach to existing approaches. First, however, let's define skylines formally.

**Definition 2** *The* skyline *of a table $R$ over preference attributes $\mathcal{B}$, denoted by $SKY(R, \mathcal{B})$, is the set of tuples from $R$ defined as follows:*

$$SKY(R, \mathcal{B}) = \{t \in R \mid \nexists s \in R: \; s \leq t\}$$

In other words, the skyline of $R$ is the set of non-dominated tuples of $R$.

As customary, given a query $q$ and a relation $R$, we will let $ans(q, R)$ stand for the *answer of $q$ over $R$,* that is the set of tuples obtained by asking query $q$ against relation $R$. Moreover, we define a *skyline query* of $R$ over preference attributes $\mathcal{B}$, to be a query $q(R, \mathcal{B})$ over $R$ whose answer is a skyline of $R$ over $\mathcal{B}$, that is:

$$ans(q(R, \mathcal{B}), R) = SKY(R, \mathcal{B})$$

Skyline queries will play a central role in our approach to skyline computation.

## 3 Computing the skyline of a relational table

In this Section, we present an algorithm for computing the skyline of a given table $R$. Our algorithm obtains the skyline by constructing a skyline query. To find the skyline query, our algorithm traverses part of the query lattice and collects a set of non-dominated queries whose disjunction is the skyline query.

In contrast to existing methods that actually construct the skyline, and as such are sensitive to the ways the table $R$ is implemented or how its tuples are accessed, our algorithm obtains the skyline while making no assumptions on how the relation $R$ is accessed by the system while processing the skyline query.

Our algorithm uses the notion of *successors* of a query, defined as follows. First, for each preference attribute $B_i$ and value $b_i$ in the domain of $B_i$, let's denote as $succ(b_i)$ the successor of $b_i$ in the linear order of the domain of $B_i$. For instance, let $B_i$ be the Hotel Rating attribute of our earlier example, having as domain the interval [1,5]. As this interval is linearly ordered by the $<$ relation, we have $succ(1) = 2$, $succ(2) = 3$, and so on. This makes $succ$ a partial function over the domain of $B_i$, undefined for the maximum $M_i$. Now, we extend the $succ$ function to preference queries as follows.

Let $q$ be a preference query $q = \langle b_1, \ldots, b_k \rangle$ such that $q \neq \bot$. This means that $b_j \neq M_j$ for at least one $j \in [1, k]$, where $M_j$ is the maximum value in the active domain of $B_j$. With no loss of generality, we shall assume that the $m$ values of $q$ that are not maximal, where $1 \leq m \leq k$, occur in the first $m$ positions of $q$, (*i.e.*, $q = \langle b_1, \ldots, b_m, M_{m+1}, \ldots, M_k \rangle$). Then, the *successors of q, succ*$(q)$, is defined to be the set of queries $succ(q) = \{q_1, \ldots, q_m\}$, such that, for all $1 \leq i \leq m$,

$$q_i = \langle b_1, \ldots, b_{i-1}, succ(b_i), b_{i+1}, \ldots, b_m, M_{m+1}, \ldots, M_k \rangle$$

Clearly, the *succ* function is undefined on the bottom of the lattice $\bot$. The following Lemma gives two important properties of the successors of $q$ for the establishment of the correctness of the following algorithm for computing a skyline query of the table $R$.

**Lemma 1.** *Let $q$ be a preference query. Then, for each query $q' \in succ(q)$ :*

1. *$q \leq q'$*
2. *there is no query $q''$ such that $q < q'' < q'$.*

We are now ready to give the algorithm for computing a skyline query of a table $R$.

**Algorithm** Skyline Query over a Single Table ($SQST$)

*Input* A non-empty table $R$, a non-empty set $\mathcal{B} = \{B_1, \ldots, B_k\}$ of preference attributes in $R$, and the projections of $R$ over $B_1, \ldots, B_k$.

*Data* We use the following variables for accumulating data during the execution of the algorithm:

– The variable $F$ is a set variable called *frontier;* it is initialized to empty, and it is used to accumulate the queries whose disjunction will be the result of the algorithm (*i.e.*, whose disjunction will be the skyline query).
– The variable $C$ is a set variable containing the set of all current candidate queries; it is initialized to the top $\top$ of the query lattice.
– The variable $S$ is a set variable used to accumulate successors of current candidate queries.
– The variable $C'$ is a (auxiliary) set variable for accumulating candidate queries for the next while-loop iteration in the algorithm; it is initialized to empty at the beginning of each iteration.

*Output* A set of preference queries over $R$, whose disjunction is a skyline query.

*Method*

```
$C \leftarrow \{\top\}; F \leftarrow \emptyset$
**while** $C \neq \emptyset$ **do**
    **for all** $c \in C$ such that $ans(c, R) \neq \emptyset$ **do**
        $C \leftarrow C \backslash \{c\}; F \leftarrow F \cup \{c\}$
        end for
        $C' \leftarrow \emptyset$
        for all $c \in C$ do
            for all $s \in succ(c)$ such that no query in $F$ Pareto dominates $s$ do
                $C' \leftarrow C' \cup \{s\}$
            end for
        end for
        $C \leftarrow C'$
    end while
    return $F$
```

Informally, the algorithm works as follows:

- If the top query $\top$ of the lattice is non-empty (*i.e.*if its answer over $R$ is non-empty) then the algorithm terminates; $\top$ is the output of the algorithm, and the answer of $\top$ over $R$ is the skyline.
- Otherwise, each successor query $c$ of $\top$ might be a candidate for contributing to the skyline, and this is checked as follows:
  - if the query $c$ is non-empty then it is added to $F$ (*i.e.*to the variable that accumulates the queries contributing to the skyline);
  - otherwise, each successor of $c$ that is not dominated by a query in $F$ becomes a candidate, by being added to the variable $C'$ (*i.e.*to the variable that accumulates all candidate queries for the next iteration). The so selected successors are finally transferred to the variable $C$.

This process is repeated until there is no more candidate left (*i.e.*until $C$ is empty).

Upon termination of the algorithm, $F$ contains all queries $q$ such that: (a) the answer to $q$ is non-empty, and (b) $q$ is not dominated by another query. The disjunction of all queries in $F$ is then the skyline query, and the answer of the skyline query over $R$ is the skyline of $R$.

$R$

| HID | Price | Rating |
|---|---|---|
| $h_1$ | 200 | 2 |
| $h_2$ | 150 | 5 |
| $h_3$ | 100 | 3 |
| $h_4$ | 300 | 2 |
| $h_5$ | 350 | 1 |
| $h_6$ | 100 | 8 |

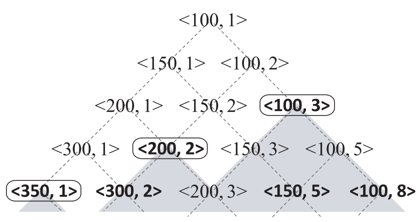

**Fig. 1.** Example relation and query lattice.

Let us illustrate the algorithm by using the example table $R$ in Fig 1. We assume the preference attributes to be *Price* and *Rating* and the preference to be $Price \to min$ and $Rating \to min$. The projections of $R$ over $Price$ and $Rating$, sorted in ascending order, are as follows:

– $Price : 100, 150, 200, 300, 350$
– $Rating : 1, 2, 3, 5, 8$

The diagram in Fig 1 shows a part of the query lattice derived from $R$. In this diagram, queries having non-empty answers are emphasized by bold letters. Queries contributing to the skyline are enclosed by rounded rectangles. Gray triangles in the diagram represent areas dominated by queries in the skyline. As we can see in the diagram, the top of the query lattice is $\top = \langle 100, 1\rangle$. Therefore, we start the first iteration of the algorithm with $C = \{\langle 100, 1\rangle\}$ and $F = \emptyset$. At the end of each subsequent iteration of the while-loop, the contents of $C$ and $F$ change as follows:

– end of 1st iteration: $C = \{\langle 150, 1\rangle, \langle 100, 2\rangle\}, F = \emptyset$
– end of 2nd iteration: $C = \{\langle 200, 1\rangle, \langle 150, 2\rangle, \langle 100, 3\rangle\}, F = \emptyset$

In the third iteration, the query $\langle 100, 3\rangle \in C$ has a non-empty result therefore $\langle 100, 3\rangle$ leaves $C$ and enters $F$. We then consider the successors of the queries left in $C$. $\langle 150, 3\rangle \in succ(\langle 150, 2\rangle)$ is dominated by $\langle 100, 3\rangle \in F$ therefore $\langle 150, 3\rangle$ is omitted from $C'$, which accumulates candidate queries for the next iteration.

– end of 3rd iteration: $C = \{\langle 300, 1\rangle, \langle 200, 2\rangle\}, F = \{\langle 100, 3\rangle\}$
– end of 4th iteration: $C = \{\langle 350, 1\rangle\}, F = \{\langle 100, 3\rangle, \langle 200, 2\rangle\}$
– end of 5th iteration: $C = \emptyset, F = \{\langle 100, 3\rangle, \langle 200, 2\rangle, \langle 350, 1\rangle\}$
  (the algorithm stops here)

The correctness of the algorithm is easily established by observing that the algorithm explores the lattice completely (this is guaranteed by the second property of the *succ* function in the above Lemma), retaining only maximal queries (this is guaranteed by the test on dominance performed by the algorithm and also by the fact that the successors of a query are all dominated by it, as stated by the first property of the *succ* function in the above Lemma).

Formally, we will denote as $SQST(R, \mathcal{B})$ the result of the $SQST$ algorithm having $R$ and $\mathcal{B}$ as the input non-empty relation and preference attributes, respectively. On the basis of the above observations, we state the following:

**Proposition 1** *For every relation $R$ and preference attributes $\mathcal{B}$ over $R$,*

$$ans(\bigvee SQST(R, \mathcal{B}), R) = SKY(R, \mathcal{B})$$

We note that, as all queries in $F$ are conjunctive and any two queries in $F$ differ in at least one conjunct, the answers making up the skyline actually form a partition of the skyline. This is an interesting observation when combined with the notion of rank of a query in the query lattice.

**Definition 3 (Rank of a query)** *The rank of a query $q$ in the query lattice is defined as follows: if $q$ is the root query then $rank(q) = 0$ else $rank(q)$ is the maximum length of path from the root query to $q$*

Clearly, the higher the rank of a query the less the tuples in its answer are preferred.

Now, as the skyline of $R$ is partitioned by the answers to the queries in $F$, one can ask new kinds of queries. For example, one can ask the query: "give me the best tuples from the skyline". The answer to this query will be the answer to the query of lowest rank in $F$. Similarly, one can ask the query: "give me the ranks of all tuples in the skyline". This query will return a set of ranks, thereby giving a useful information as to how far are the tuples of the skyline from the most preferred tuples. A detailed discussion of the relationship between "most preferred" and "non-dominated" tuples is given in a forthcoming paper.

## 4 Skylines of joins

We now consider the computation of the skyline over the join of two tables $R_1$ and $R_2$. To this end, we introduce the necessary concepts.

Let $R_1$ and $R_2$ be relations with $A_1^1, \ldots, A_1^{n_1}$ and $A_2^1, \ldots, A_2^{n_2}$ as attributes, respectively, and let $\mathcal{B}_i = \{B_i^1, \ldots, B_i^{k_i}\}$, $k_i \leq n_i$, be a set of attributes of $R_i$, called the *preference attributes* of $R_i$, for $i = 1, 2$.

We shall denote as $(Q_1, \leq_1)$ and $(Q_2, \leq_2)$ the query lattices over $R_1$ and $R_2$, respectively. Moreover, $\otimes_i$ and $\oplus_i$ will stand for the least upper bound and the greatest lower bound of any two preference queries over $R_i$, respectively.

Let $\mathcal{J}_i = \{J_i^1, \ldots, J_i^l\}$, be a set of attributes of $R_i$ disjoint form the preference attributes $\mathcal{B}_i$, for $i = 1, 2$. A *join* over $\mathcal{J}_1$ and $\mathcal{J}_2$ is a relational equijoin $R_1 \bowtie_e R_2$, whose join expression $e$ is given by $J_1^1 = J_2^1, \ldots, J_1^l = J_2^l$. In order to simplify the model and with no loss of generality, we will consider the join attributes to be the same for the two relations, that is $\mathcal{J}_1 = \mathcal{J}_2$, and moreover to consist of a single attribute $J$, that is $e$ is given by $J = J$.

Intuitively, a preference query over a join $R_1 \bowtie_e R_2$ is a $(k_1 + k_2)$-tuple whose first $k_1$ elements make up a query in $Q_1$ and whose last $k_2$ elements make up a query in $Q_2$. In order to simplify notation, we will commit a slight abuse and write $\langle q_1, q_2 \rangle$ to represent a preference query over $R_1 \bowtie_e R_2$, where $q_1 \in Q_1$ and $q_2 \in Q_2$. As a consequence, the set of preference queries over $R_1 \bowtie_e R_2$, that we shall denote as $Q_{\bowtie}$, is given by the Cartesian product of the set of preference queries over $R_1$ and $R_2$, that is:

$$Q_{\bowtie} = Q_1 \times Q_2$$

Now, let $\leq_{\bowtie}$ be the Pareto preference relation over $Q_{\bowtie}$. As we have already seen in the previous Section, $(Q_{\bowtie}, \leq_{\bowtie})$ is a complete lattice. Moreover, it is not difficult to see that:

**Proposition 2** *$(Q_{\bowtie}, \leq_{\bowtie})$ is the product of $(Q_1, \leq_1)$ and $(Q_2, \leq_2)$. That is, letting $q$ and $q'$ be preference queries in $Q_{\bowtie}$ such that $q = \langle q_1, q_2 \rangle$ and $q' = \langle q_1', q_2' \rangle$, where $q_1, q_1' \in Q_1$ and $q_2, q_2' \in Q_2$, we have:*

1. *$q \leq_{\bowtie} q'$ iff $q_1 \leq_1 q_1'$ and $q_2 \leq_2 q_2'$*
2. *$q \otimes_{\bowtie} q' = \langle q_1 \otimes_1 q_1', q_2 \otimes_2 q_2' \rangle$*
3. *$q \oplus_{\bowtie} q' = \langle q_1 \oplus_1 q_1', q_2 \oplus_2 q_2' \rangle$*

From now on we shall simplify notation by omitting subscripts, unless this creates ambiguity.

We shall call *join* values the set of tuples $V = X_1 \cap X_2$, where:

$$X_i = \pi_J(R_i) \quad \text{for } i = 1, 2$$

By definition of join, a tuple $t$ in $R_i$ contributes to the join if and only if its projection over the join attribute $J$ is in $V$, that is $t.J \in V$, for $i = 1, 2$. Likewise, a query $q$ in $Q_i$ *may* contribute to the skyline of the join only if it occurs in the join. For each join value $v \in V$, we define the $v-$partition of $R_i$, denoted $S_i(v)$, as follows :

$$S_i(v) = \pi_{\mathcal{B}_i}(\sigma_{J=v}(R_i)) \quad \text{for } i = 1, 2$$

In practice, each $v-$partition includes the queries that contribute to the join $R_1 \bowtie R_2$. In particular:

$$\pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie R_2) = \bigcup_{v \in V} S_1(v) \times S_2(v)$$

$v-$partitions play an important role in determining the skyline of the join $R_1 \bowtie R_2$ *without* computing the join.

**Proposition 3** *A query $\langle q_1, q_2 \rangle \in \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(SKY(R_1 \bowtie R_2, \mathcal{B}_1 \cup \mathcal{B}_2))$ iff for some join value $v \in V$, $q_i \in \pi_{\mathcal{B}_i}(SKY(S_i(v), \mathcal{B}_i))$ for $i = 1, 2$ and for no other $v' \in V$ there exists skylines $q_i' \in \pi_{\mathcal{B}_i}(SKY(S_i(v'), \mathcal{B}_i))$ such that $q_i' \leq q_i$ for $i = 1, 2$.*
*Proof: ($\rightarrow$) If for some join value $v \in V$, $q_i \in \pi_{\mathcal{B}_i}(SKY(S_i(v), \mathcal{B}_i))$ for $i = 1, 2$ then $\langle q_1, q_2 \rangle \in \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie R_2)$, and moreover for 1 in Proposition 2 there exists no other query $\langle q_1', q_2' \rangle \in \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie R_2)$ where $q_i' \in S_i(v)$ for $i = 1, 2$ such that $\langle q_1', q_2' \rangle \leq \langle q_1, q_2 \rangle$. If for no other $v' \in V$ there exists skylines $q_i' \in \pi_{\mathcal{B}_i}(SKY(S_i(v'), \mathcal{B}_i))$ such that $q_i' \leq q_i$ for $i = 1, 2$ then again from 1 in Proposition 2 there exists no $\langle q_1', q_2' \rangle$ such that $\langle q_1', q_2' \rangle \leq \langle q_1, q_2 \rangle$. Hence $\langle q_1, q_2 \rangle \in \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(SKY(R_1 \bowtie R_2, \mathcal{B}_1 \cup \mathcal{B}_2))$.*
*($\leftarrow$) Suppose not. Then, either (a) for no join value $v \in V$, $q_i \in \pi_{\mathcal{B}_i}(SKY(S_i(v), \mathcal{B}_i))$ for $i = 1, 2$ or (b) there exists a join value $v' \in V$ and skylines $q_i' \in \pi_{\mathcal{B}_i}(SKY(S_i(v'), \mathcal{B}_i))$ such that $q_i' \leq q_i$ for $i = 1, 2$. In case (a), there are two sub-cases: (a1) for no join value $v \in V$, $q_i \in S_i(v)$ for $i = 1, 2$. In this case, $\langle q_1, q_2 \rangle \notin \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie R_2)$, against the hypothesis. (a2) for some join value $v \in V$, $q_i \in S_i(v)$, but either $q_1 \notin \pi_{\mathcal{B}_1}(SKY(S_1(v), \mathcal{B}_1))$ or $q_2 \notin \pi_{\mathcal{B}_2}(SKY(S_2(v), \mathcal{B}_2))$. Then let $q_i'$ be such that $q_i' \in \pi_{\mathcal{B}_i}(SKY(S_i(v), \mathcal{B}_i))$. Such $q_i'$ are guaranteed to exist because $S_i(v)$ is finite and partially ordered by Pareto preference. By hypothesis, either $q_1' \neq q_1$ or*

*$q'_2 \neq q_2$. Then $\langle q'_1, q'_2 \rangle \in \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie R_2)$ and by 1 in Proposition 2 $\langle q'_1, q'_2 \rangle \leq \langle q_1, q_2 \rangle$, therefore $\langle q_1, q_2 \rangle \notin \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(SKY(R_1 \bowtie R_2, \mathcal{B}_1 \cup \mathcal{B}_2))$, against the hypothesis. (b) If there exists a join value $v' \in V$ and skylines $q'_i \in \pi_{\mathcal{B}_i}(SKY(S_i(v'), \mathcal{B}_i))$ such that $q'_i \leq q_i$ for $i = 1, 2$ then $\langle q'_1, q'_2 \rangle \in \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(R_1 \bowtie R_2)$ and by 1 in Proposition 2 $\langle q'_1, q'_2 \rangle \leq \langle q_1, q_2 \rangle$, therefore $\langle q_1, q_2 \rangle \notin \pi_{\mathcal{B}_1 \cup \mathcal{B}_2}(SKY(R_1 \bowtie R_2, \mathcal{B}_1 \cup \mathcal{B}_2))$, against the hypothesis.*

We now provide an algorithm for computing the skyline queries over the join of two tables $R1$ and $R_2$ (without computing the join.). The algorithm exploits Proposition 3, and uses the *SQST* algorithm for computing skylines over $v$-partitions of the given tables. As a result, the algorithm returns two sets of queries, one over $R_1$, the other over $R_2$, for selecting from each table the tuples that will generate the skyline of the join $R_1 \bowtie R_2$, and only those.

**Algorithm** The Skyline Queries over a Join (*SQJ*)

*Input* Non-empty relations $R_1$ and $R_2$, non-empty sets $\mathcal{B}_1$ and $\mathcal{B}_2$ of preference attributes in $R_1$ and $R_2$, respectively, and the join values $V$.

*Data* $G$ is a set variable initialized to empty and used to accumulate all candidate skyline queries, resulting from the Cartesian products of the skyline queries over the same join value. $R$ is a set variable where the skyline of $G$ is computed. $P_i$ and $F_i$ (for $i = 1, 2$) are auxiliary set variables, used to store $v$-partitions and final results, respectively.

*Output* Two sets of queries, one over $R_1$ and the other over $R_2$.

*Method*
```
  $G \leftarrow \emptyset$
  for all $v \in V$ do
    $P_1 \leftarrow SQST(S_1(v), \mathcal{B}_1)$
    $P_2 \leftarrow SQST(S_2(v), \mathcal{B}_2)$
    $G \leftarrow G \cup (P_1 \times P_2 \times \{v\})$
  end for
  $R \leftarrow SQST(G, \mathcal{B}_1 \cup \mathcal{B}_2)$
  $F_1 \leftarrow \pi_{\mathcal{B}_1 \cup \{J\}}(R)$
  $F_2 \leftarrow \pi_{\mathcal{B}_2 \cup \{J\}}(R)$
  return $F1$, $F_2$
```

We note that the algorithm operates in three passes:

1. In the first pass, it gathers (in $G$) all candidate results by looping over all join values. This is in fact required by the first condition of Proposition 3, which states that $\langle q_1, q_2 \rangle$ is in a skyline query of the join if both $q_1$ and $q_2$ are skyline queries over the $v$-partitions for the same join value $v$.
2. In the second pass, it removes from $G$ the compound queries that are dominated by some other compound query, as required by the second condition of Proposition 3.

3. Finally, it slices the compound queries vertically, in order to obtain queries over $R_1$ (these are stored in $F_1$) and queries over $R_2$ (in $F_2$). Notice that the join attribute $J$ must be transferred all along in order to generate the correct queries.

Clearly, there is no other way of proceeding since it is necessary to obtain *all* candidate queries in order to apply the second condition of Proposition 3.

Formally, we will denote as $SQJ(R_1, R_2, \mathcal{B}_1, \mathcal{B}_2)_i$ the $i-$th result of the $SQJ$ algorithm having $R_1$ and $R_2$ as the input non-empty relations and $\mathcal{B}_1$ and $\mathcal{B}_2$ as preference attributes, respectively. For readability, we will abbreviate $SQJ(R_1, R_2, \mathcal{B}_1, \mathcal{B}_2)_i$ as $SQJ_i$.

On the basis of the above observations and of Proposition 3, we therefore state the following:

**Proposition 4** *For every pair of relations $R_1$ and $R_2$ and preference attributes $\mathcal{B}_1$ and $\mathcal{B}_2$ over them,*

$$SQST(ans(\bigvee SQJ_1, R_1) \bowtie ans(\bigvee SQJ_2, R_2), \mathcal{B}_1 \cup \mathcal{B}_2) = SKY(R_1 \bowtie R_2, \mathcal{B}_1 \cup \mathcal{B}_2)$$

Let us demonstrate the SQJ algorithm by using table $R_1$ (hotels) and $R_2$ (restaurants) in Fig. 2. Suppose we want to find best combinations of hotels and restaurants in the same “Location”, by minimizing “Price”, “Rating”, “Distance” and “Location” (we took this example from [23]). In this case, the preference attributes are $\mathcal{B}_1 = \{Price, Rating\}$, $\mathcal{B}_2 = \{Distance, Ranking\}$ and the join attribute is $\mathcal{J} = \{Location\}$. From the intersection of values appearing in the *Location* attribute in $R_1$ and $R_2$, we can obtain join values $V = \{A, B, C\}$.

In the algorithm, firstly we gather candidate skyline queries for each join value. In the example, for a join value $A \in V$, we obtain v-partition $S_1(A)$ and $S_2(A)$. Then we compute skylines $P_1, P_2$ (emphasized in tables $S_1$, $S_2$ by bold letters) from $S_1(A)$, $S_2(A)$ by the applying the SQST algorithm. Finally we make a Cartesian product $P_1 \times P_2 \times \{A\}$ and append it to $G$ that accumulates candidate skyline queries.

After iterating this candidate gathering process for every join value, we apply SQST to $G$ to obtain the skyline (emphasized in tables $G$ by bold letters) over the join $R_1 \bowtie R_2$. It is important to note the difference of size between $R_1 \bowtie R_2$ and $G$. In this example, $R_1 \bowtie R_2$ produces 12 tuples but $G$ contains 8 tuples. Therefore, we can compute the skyline from $G$ with less cost than by computing directly from $R_1 \bowtie R_2$.

In the example, $\langle h_5, r_5 \rangle$ is not a skyline in the join because for the join value $B$, we have $h_2$ dominating $h_5$ and $r_1$ dominating $r_5$. On the other hand, neither $h_6$ nor $r_4$ are skyline in their table, but they form a skyline in the join because they are skylines in their join group and there is no other group in which both are dominated ($h_6$ is dominated by a query $h_3$ in the $A$ group, whereas $r_4$ is dominated by $r_2$ in the $C$ group, and there is no single group in which both are dominated.)

***$R_1$(Hotels)***

| HID | Price | Rating | Location |
|---|---|---|---|
| $h_1$ | 100 | 8 | A |
| $h_2$ | 150 | 5 | B |
| $h_3$ | 200 | 1 | A |
| $h_4$ | 400 | 2 | A |
| $h_5$ | 300 | 7 | C |
| $h_6$ | 350 | 3 | B |

***$S_1$***

| | Price | Rating | |
|---|---|---|---|
| $S_1(A)$ | **100** | **8** | $P_1^A = \{\langle 100, 8\rangle, \langle 200, 1\rangle\}$ |
| | **200** | **1** | |
| | 400 | 2 | |
| $S_1(B)$ | **150** | **5** | $P_1^B = \{\langle 150, 5\rangle, \langle 350, 3\rangle\}$ |
| | **350** | **3** | |
| $S_1(C)$ | **300** | **7** | $P_1^C = \{\langle 300, 7\rangle\}$ |

***$R_2$(Restaurants)***

| RID | Distance | Ranking | Location |
|---|---|---|---|
| $r_1$ | 150 | 4 | B |
| $r_2$ | 250 | 2 | C |
| $r_3$ | 500 | 1 | A |
| $r_4$ | 400 | 3 | B |
| $r_5$ | 200 | 5 | C |
| $r_6$ | 500 | 6 | A |

***$S_2$***

| | Distance | Ranking | |
|---|---|---|---|
| $S_2(A)$ | **500** | **1** | $P_2^A = \{\langle 500, 1\rangle\}$ |
| | 500 | 6 | |
| $S_2(B)$ | **150** | **4** | $P_2^B = \{\langle 150, 4\rangle, \langle 400, 3\rangle\}$ |
| | **400** | **3** | |
| $S_2(C)$ | **250** | **2** | $P_2^C = \{\langle 250, 2\rangle, \langle 200, 5\rangle\}$ |
| | **200** | **5** | |

***$G$***

| Price | Rating | Distance | Ranking | Location | |
|---|---|---|---|---|---|
| **100** | **8** | **500** | **1** | A | $P_1^A \times P_2^A \times \{A\}$ |
| **200** | **1** | **500** | **1** | A | |
| **150** | **5** | **150** | **4** | B | $P_1^B \times P_2^B \times \{B\}$ |
| **150** | **5** | **400** | **3** | B | |
| **350** | **3** | **150** | **4** | B | |
| **350** | **3** | **400** | **3** | B | |
| **300** | **7** | **250** | **2** | C | $P_1^C \times P_2^C \times \{C\}$ |
| 300 | 7 | 200 | 5 | C | |

***$F_1$***

| Price | Rating | Location |
|---|---|---|
| 100 | 8 | A |
| 200 | 1 | A |
| 150 | 5 | B |
| 350 | 3 | B |
| 300 | 7 | C |

***$F_2$***

| Distance | Ranking | Location |
|---|---|---|
| 500 | 1 | A |
| 150 | 4 | B |
| 400 | 3 | B |
| 250 | 2 | C |

**Fig. 2.** An example of skyline over the join of two tables

## 5 Concluding Remarks

We have seen a novel approach to computing the skyline of a relational table with respect to preferences expressed over one or more attributes with ordered domains. Our approach is based on what we called the query lattice of the table, and our basic algorithm constructs the skyline as the union of the answers to a subset of queries from that lattice — hence *without* directly accessing the table $R$. Therefore, in contrast to all existing techniques, our approach is independent of how the table $R$ is implemented or how its tuples are indexed. We have demonstrated the generality of our approach by computing the skyline over the join of two tables based on the product of their individual query lattices — therefore *without* performing the join.

We note that our method is applicable to a computational geometry setting as well. Indeed, a discrete (finite) $n$-dimensional Euclidean space $S$ can be thought of as a relational table $T(S)$ in which: (a) the attributes are the $n$ dimensions of $S$ and (b) each tuple of $T(S)$ represents a point in $S$. Moreover, the answer to a query $q = \langle b_1, \ldots, b_k \rangle$ from the query lattice of $T(S)$ is the set of all points of $S$ such that $(B_1 = b_1) \wedge \ldots \wedge (B_k = b_k)$, where $B_1, \ldots, B_k$ are the corresponding dimensions; in other words, the answer to $q$ is the set of points having the same coordinate values over the dimensions $B_1, \ldots, B_k$. Additionally, our method can be applied to the Cartesian product of two or more spaces through the product lattice of the individual spaces. We are currently pursuing work in two different directions, namely refining skyline analysis and applying our approach to a distributed setting:

1. *Refining skyline analysis*
   As we mentioned in section 3, the skyline query returned by our basic algorithm is the disjunction of a set of queries from the query lattice, say $q_1 \wedge \ldots \wedge q_m$; and the answers to these queries actually partition the skyline into disjoint subsets. Moreover, these queries have different ranks, in general. Therefore it now becomes possible to ask finer queries regarding the skyline such as "give me the best tuples of the skyline" (meaning the answer to the query $q_i$ of highest rank), or "return the skyline by presenting the answers to $q_i$'s in increasing order of rank", and so on. In this respect, we would also like to investigate in more detail the relationship between "most preferred tuple" and "non-dominated tuple".
2. *Applying our method to a distributed setting*
   As information systems store bigger and bigger volumes of data today, data management and in particular query processing presents difficult challenges. From the user viewpoint, big volumes of data imply answers of large size. By returning the best tuples (in terms of user preferences), the skyline query relieves the user from having to deal with answers of large size; and having the possibility to further refine the skyline (as mentioned above) further contributes in that direction. However, in recent years, data management and data storage have become increasingly distributed, and distribution presents additional challenges for query processing. Adapting the skyline operator to a distributed setting is one of the research lines that we are currently pursuing. We believe that our approach to skyline computation through query lattices is particularly well suited in a distributed environment, where computation can be distributed and recomposed in the form of the product lattice.

## References

1. I. Bartolini, P. Ciaccia, and M. Patella. SaLSa: computing the skyline without scanning the whole sky. In Proceedings of CIKM, 2006.
2. I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. ACM Trans. Database Syst., 33(4), 2008.
3. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In Proceedings of ICDE, pages 421-430, 2001.
4. C.-Y. Chan, P.-K. Eng, K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In Proc. of SIGMOD 2005, pp. 03–214, 2008.

5. J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In Proceedings of ICDE, pages 717-816, 2003.
6. B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, Y. Zhou. Parallel Distributed Processing of Constrained Skyline Queries by Filtering. In Proc. of ICDE 2008, pp. 546-555, 2008.
7. P. Godfrey, R. Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. The VLDB Journal, vol 16(1), pp. 5–28, 2007.
8. W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In Proceedings of ICDE, 2007.
9. W. Jin, M. Morse, J. Patel, M. Ester, and Z. Hu. Evaluating skylines in the presence of equi-joins. In Proc. of ICDE, 2010.
10. N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In Proceedings of VLDB, 2006.
11. H.T. Kung, F. Luccio, F.P. Preparata. On finding the maxima of a set of vectors. Journal of the ACM, vol. 22(4), pp. 469–476, 1975.
12. M.E. Khalefa, M.F. Mokbel, J.J. Levandoski. Skyline Query Processing for Incomplete Data. In Proc. of ICDE 2008, pp. 556-565, 2008.
13. D. Kossmann , F. Ramsak , S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In Proc. of VLDB 2002, pp. 275–286, 2002.
14. X. Lin , Y. Yuan , W. Wangnicta , S. Wales. Stabbing the sky: Efficient skyline computation over sliding windows. In Proc. of ICDE 2005, pp. 502–513, 2005.
15. M. Morse , J. Patel , H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In Proc. of VLDB 2007, pp. 267–278, 2007.
16. J. Pei , W. Jin , M. Ester , Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In Proc. of VLDB 2005, pp. 253–264, 2005.
17. J. Pei , B. Jiang , X. Lin , Y. Yuan. Probabilistic skylines on uncertain data. In Proc. of VLDB 2007, pp. 15–26, 2007.
18. F.P. Preparata, M.I. Shamos. Computational Geometry Computational Geometry, Springer-Verlag, 1985.
19. D. Papadias, Yufei Tao, Greg Fu, Bernhard Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In Proc. of SIGMOD 2003, pp. 467–478, 2003.
20. V. Raghavan, E. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In Proceedings of ICDE, 2010.
21. K.-L. Tan, P.-K. Eng, B.C. Ooi. Efficient Progressive Skyline Computation. In Proc. of VLDB 2001, pp. 301-310, 2001.
22. Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In Proceedings of ICDE, 2006.
23. A. Vlachou, C. Doulkeridis, N. Polyzotis. Skyline query processing over joins. In Proceedings of SIGMOD 2011, pp. 73–84, 2011.
24. A. Vlachou, K. Nørvåg. Bandwidth-constrained distributed skyline computation. In Proc. of MobiDE 2009, pp. 17–24, 2009.
25. D. Sun, S. Wu, J. Li, and A. K. H. Tung. Skyline-join in distributed databases. In ICDE Workshops, 2008.
26. Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. PVLDB, vol. 2(1), 898-909, 2009.
27. P. Wu , C. Zhang , Y. Feng , B.Y. Zhao , D. Agrawal , A.E. Abbadi. Parallelizing skyline queries for scalable distribution. In Proc. of EDBT 2006, pp. 112-130, 2006.
28. Y. Yuan , X. Lin , Q. Liu , W. Wang , J.X. Yu , Q. Zhang. Efficient Computation of the Skyline Cube. In Proc. of VLDB 2005, pp. 241–252, 2005.

29. S. Zhang , N. Mamoulis , S.W. Cheung. Scalable skyline computation using object-based space partitioning. In Proc. of SIGMOD 2009, pp. 483-494, 2009.