SaLSa: Computing the Skyline without Scanning the Whole Sky[*]

Ilaria Bartolini, Paolo Ciaccia, Marco Patella
DEIS, University of Bologna, Italy
{ibartolini,pciaccia,mpatella}@deis.unibo.it

ABSTRACT

Skyline queries compute the set of Pareto-optimal tuples in a relation, i.e., those tuples that are not *dominated* by any other tuple in the same relation. Although several algorithms have been proposed for efficiently evaluating skyline queries, they either require to extend the relational server with specialized access methods (which is not always feasible) or have to perform the dominance tests on *all* the tuples in order to determine the result. In this paper we introduce SaLSa (*Sort and Limit Skyline algorithm*), which exploits the sorting machinery of a relational engine to order tuples so that only a subset of them needs to be examined for computing the skyline result. This makes SaLSa particularly attractive when skyline queries are executed on top of systems that do not understand skyline semantics or when the skyline logic runs on clients with limited power and/or bandwidth.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Skyline, Monotone functions, Client/Server architecture

1. INTRODUCTION

The *skyline* of a relation r is the set of Pareto-optimal, or *undominated*, tuples in r. According to the Pareto principle, a tuple p dominates another tuple p_i if p is at least as good as p_i on all the skyline attributes and strictly better than p_i on at least one attribute.

Example 1 *Consider the relation Hotels(Name, Price, Rating, Location) and the instance:*

Name	*Price*	*Rating*	*Location*
Jolly	*$ 30*	*2*	*sea*
Continental	*$ 35*	*2*	*mountain*
Excelsior	*$ 60*	*3*	*city*
Rome	*$ 60*	*5*	*sea*
Holiday	*$ 50*	*4*	*country*
Capri	*$ 60*	*4*	*city*

A skyline query over the attributes Price (to be minimized), Rating (to be maximized), and Location, for which the most preferred value is 'sea' whereas 'city' is the worst, will return hotels Jolly, Rome, and Holiday, since Jolly dominates Continental (better price, same rating, better location) and both Rome and Holiday dominate Excelsior and Capri. Using Preference SQL [11], this query can be expressed as:

```
SELECT *
FROM Hotels
PREFERRING LOWEST(Price) AND HIGHEST(Rating)
AND Location='sea' ELSE Location<>'city' □
```

Skyline queries are a specific, yet relevant, example of preference queries [6, 10], and have been recognized as a useful and practical way to make database systems more flexible in supporting user requirements [5]. Consequently, there has been a growing interest in algorithms for efficiently evaluating skyline queries on large databases [4, 8, 13, 9]. With respect to such works, which concentrate on how to implement the skyline operator *within* a database system, in this paper we take a slightly different perspective on the problem, by addressing the following important issue: *how can skyline queries be efficiently computed if the underlying data server has no knowledge at all of the skyline logic?* In other terms, how can an application, running *on top* of a standard database system, evaluate a skyline query? Note that this is precisely the problem that one has to face when using *any* currently available commercial database system. The same is true if one wants to provide skyline functionality on top of a web-accessible data source.

In scenarios like these the only currently available alternative is to run a skyline algorithm on the client side, and compute the result by requesting to the underlying data source *all* the data in the target relation. For instance, computing the skyline query in Example 1 would require the client to fetch from the server holding the Hotels relation *all* its (presumably many) hotels. It should be observed that, no matter how smart the skyline algorithm is designed, this strategy will pay an excessive communication overhead,

[*]This work is partially supported by the MIUR PRIN Project WISDOM (Web Intelligent Search based on DOMain ontologies).

CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.
Copyright 2006 ACM 1-59593-433-2/06/0011 ...$5.00.

which might nullify any effort spent in the implementation of the skyline algorithm. This is particularly true for clients with a limited bandwidth connection, as the following example illustrates.

Example 2 *Consider a skyline query issued by a wireless client (e.g., a PDA) on a 802.11b network. If the relation has 20,000 tuples, each of 100 bytes, and the effective data transfer rate between the client and the server is 1 Mbps (1 Megabits per second),*[1] *then shipping all the tuples to the client will require about 15 seconds. On the other hand, state-of-the-art algorithms can compute a 6-dimensional skyline on such relation in less than 1 second, i.e., 1 order of magnitude faster (see also our experiments in Section 5).* □

Motivated by above observations, in this paper we introduce a novel algorithm, called SaLSa (*Sort and Limit Skyline algorithm*). SaLSa takes from the SFS (*Sort Filter Skyline*) algorithm by Chomicki et al. [8] the idea of *pre-sorting* the input relation before running the filter step in which dominance tests are executed. Thus, both SFS and SaLSa need to perform a topological sort of the input data, and as such both can *progressively* return undominated tuples as soon as they discover them. However, while for SFS a "good" sorting function is one that puts in the first positions those tuples that are likely to dominate many other tuples, thus leading to reduce the number of dominance tests, sorting data in SaLSa is mainly used as a means to stop fetching tuples from the input stream, thus effectively *limiting* the number of tuples to be read. In other terms, SaLSa relies on sorting functions that can guarantee that *all* tuples beyond a certain point in the input stream are dominated by some already seen tuple. As an example, SaLSa is able to determine the 10 tuples of a 3-dimensional skyline over the NBA dataset by reading only 3783 tuples out of 17791, thus saving 79% of the transfer cost (see Section 5 for description of datasets and details on experiments). With numbers as in Example 2, this would save about 11 seconds of transfer time.

The rest of the paper is organized as follows. In Section 2 we review the semantics of a skyline query and algorithms for its evaluation. Section 3 introduces the SaLSa algorithm. Section 4 provides details on some implementation issues. Section 5 presents experimental results on both real and synthetic datasets, and Section 6 concludes.

2. SKYLINE ALGORITHMS

This section briefly reviews the definition of skyline and existing algorithms for its evaluation.

Let $R(A_1, \ldots, A_d)$ be a relation schema, and let $dom(A_j)$ be the domain of attribute A_j. Further, let $\mathcal{D} = dom(A_1) \times \ldots \times dom(A_d)$. Let r be a relation over R, i.e., a set of tuples, or d-dimensional *points*, from $\mathcal{D}$. In the following, the terms tuple and point will be used interchangeably.

The *skyline* of r, $\mathcal{S} = \mathcal{S}(r)$, is the subset of points in r that are Pareto-optimal.[2] A point p is Pareto-optimal, or *undominated*, iff there is no other point $p_i \in r$ that is not worse than p on all attributes (or *coordinates*) and strictly better than p on at least one attribute. Assuming without loss of generality that on each attribute higher values are better, and writing $p \succ p_i$ to mean that p dominates p_i, we have:

$$\mathcal{S}(r) = \{p \in r : \nexists p_i \in r : p_i \succ p\} \tag{1}$$

where:

$$p \succ p_i \Leftrightarrow \left(\bigwedge_{j=1}^{d} p[j] \geq p_i[j]\right) \wedge \left(\bigvee_{j=1}^{d} p[j] > p_i[j]\right) \tag{2}$$

and $p[j] \equiv p.A_j$ is the value of p for attribute A_j. Extending the definition to the case of categorical attributes (e.g., Location) is an easy task, which will be discussed in Section 4.

Computing the skyline is equivalent to determine the *maxima* of a set of vectors, a well-known problem in computational geometry [14]. However, algorithms developed in that field cannot be directly applied in a database scenario, since they do not take into account main memory limitations. This was observed in [4], where a *divide and conquer* algorithm, D&C, suitable for external memory was proposed. However, a recent analysis [9] shows that the average performance of D&C deteriorates with increasing skyline dimensionality, d. In [4] another algorithm, called *Block Nested Loops* (BNL), is proposed. BNL allocates in main memory a *window* W, and sequentially scans the input relation. When a point p is read, it is compared to points in W. If p is dominated by a point in W, then p is discarded, otherwise p is inserted in W. If p dominates some points in W, these are removed from W. In case the window saturates, a temporary file is used to store points that cannot be placed in W. This file is used as the input to the next pass. Eventually the algorithm terminates, since at the end of each pass the size of the temporary file can only decrease.

The SFS (*Sort Filter Skyline*) algorithm [8] improves over BNL by first sorting the input data according to decreasing values of a *monotone* function $\mathcal{F}$. This guarantees that if $\mathcal{F}(p) \geq \mathcal{F}(p_i)$, then p_i will not dominate p, written $p_i \not\succ p$. In other terms, using a monotone function corresponds to perform a *topological sort* with respect to the Pareto dominance criterion. Similarly to BNL, SFS keeps in W the undominated points seen so far. However, the monotonicity of $\mathcal{F}$ now guarantees that in the filter phase a new point p_i will never dominate an already seen point p, thus a point will never be dropped from the window. This leads to three major improvements with respect to BNL: 1) the management of W largely simplifies, 2) points in the skyline can be *progressively* returned without having to wait for all the input to be read, and 3) the number of passes of the filter phase is optimal, i.e., $\lceil |\mathcal{S}|/|W| \rceil$. From the last observation it follows that, even for moderately large skylines, SFS will likely complete in a single pass. Experimental results in [8] indeed show that SFS runs (much) faster than BNL, and that it executes less dominance tests. In particular, this is achieved by sorting data using their "volume" or, equivalently, their "entropy" (see Section 3.1.2 for details).

LESS [9] is a recent improvement of SFS that integrates in the first step of a standard external sort-merge algorithm an *elimination-filter* window, so as to earlier discarding some dominated tuples. Further, LESS combines the last merge pass of the sorting algorithm with the first skyline-filter pass.

[1] The nominal 802.11b bandwidth is 11Mbps, but this assumes no traffic at all in the network. An effective 1Mbps transfer rate is a realistic estimate for the case of moderate traffic.

[2] For simplicity we consider that all attributes are involved in the skyline. Generalization to the case where R also includes non-skyline attributes is immediate.

Although results in [9] show that LESS consistently outperforms SFS, LESS would not be applicable in scenarios in which one has no direct control on the data server (thus on the algorithm used to sort tuples). For the same reason, algorithms like NN [12] and BBS [13] that rely on the existence of specific access structures are not relevant here.

Even if not directly fitting the scenario considered in this paper, it is worth mentioning skyline algorithms based on a *distributed* access model [1, 3, 2]. Such algorithms work by querying d independent subsystems, each managing a specific skyline attribute and returning objects ordered according to the preference on that attribute (e.g., minimize the price). By iterating on the streams of incoming results, the skyline can be computed by just looking at those objects that are returned by at least one subsystem before a single object p is returned by *all* subsystems. To perform all the necessary dominance tests, thus to eventually determine the actual skyline, missing attribute values for all candidate objects are then obtained through a series of *random accesses*, each having as input the object identifier for which attribute values are sought.

Although distributed skyline algorithms can indeed limit the number of tuples to be transmitted from a server to a client, applying them in a scenario where all skyline attributes are managed by a single server would unnecessarily incur a high overhead. First, the client should issue d independent sub-queries, for each of which the server should perform a distinct sort of the input relation. Further, the client should waste resources to join the results arriving from the sub-queries. Finally, many other queries would be needed for performing the random accesses, i.e., to retrieve missing attribute values.

Considering all the above, and reminding that only algorithms that can be run on top of standard database systems are of interest in our scenario, it is evident that the best currently available alternative is indeed SFS.

3. THE SALSA ALGORITHM

The algorithm we introduce, called SaLSa (for *Sort and Limit Skyline algorithm*), builds on the basic idea that, if the input relation r is sorted according to a suitably chosen monotone function, then it is possible to determine the skyline of r *without applying the skyline filter to all the points.* In general, this might drastically reduce the number of tuples to be read and, depending on the specific instance and sorting function, it might reduce the number of dominance tests as well. Since SaLSa shares with SFS the idea of presorting the input relation, it also keeps all the SFS strengths: simplified management of the window, incremental delivery of results, and optimal number of passes of the filter phase.

We illustrate how SaLSa works when a single pass is sufficient to complete the evaluation. Extension to the case where skyline size exceeds the available main memory is managed as in SFS, and not reported here for brevity.

SaLSa starts by initializing to r the set $\mathcal{U}$ of *unread tuples*. It also makes use of a *stop point*, p_{stop}, which is used to earlier terminate reading tuples. Step 2 sorts $\mathcal{U}$ according to decreasing values of a monotone function $\mathcal{F}$. This is actually done by issuing the following standard, i.e., non-skyline, SQL query to the server:

```
SELECT *
FROM $R$
ORDER BY $\mathcal{F}$ DESC
```

Each time a new point p is read from $\mathcal{U}$, p is compared against the current skyline $\mathcal{S}$. If none of the points in $\mathcal{S}$ dominates p (i.e., $\mathcal{S} \not\succ p$), p is inserted into $\mathcal{S}$. This might possibly trigger the update of the stop point (step 5). At step 6 SaLSa checks if it has gained sufficient evidence to conclude that no further point in $\mathcal{U}$ can be part of the skyline, i.e., all points in $\mathcal{U}$ are dominated by p_{stop} ($p_{stop} \succ \mathcal{U}$). If this is so, the algorithm terminates.

Algorithm 1 SaLSa

```
1: $\mathcal{S} \leftarrow \emptyset, \mathcal{U} \leftarrow r$, $stop \leftarrow false$, $p_{stop} \leftarrow$ undefined
2: sort $\mathcal{U}$ according to $\mathcal{F}$
3: while not $stop \land \mathcal{U} \neq \emptyset$ do
4:     $p \leftarrow$ get next point from $\mathcal{U}$, $\mathcal{U} \leftarrow \mathcal{U} \setminus \{p\}$
5:     if $\mathcal{S} \not\succ p$ then $\mathcal{S} \leftarrow \mathcal{S} \cup \{p\}$, update $p_{stop}$
6:     if $p_{stop} \succ \mathcal{U}$ then $stop \leftarrow true$
7: return $\mathcal{S}$
```

Two key factors ultimately determine the actual performance of SaLSa: the choice of the sorting function, which might severely influence the number of tuples to be read, and the strategy for choosing the stop point. Before analyzing both issues, we introduce some basic terminology.

Assume that during the execution of SaLSa the last point p read so far has value $\mathcal{F}(p) = l$. We say that $\mathcal{F}$ is at *level* l after reading p, and denote with $l(\mathcal{F}, r)$ the level at which SaLSa eventually halts, also called the *stop level of* $\mathcal{F}$ *on* r. Similarly, $\mathcal{U}(\mathcal{F}, r)$ denotes the value of $\mathcal{U}$ when the algorithm terminates.

When $\mathcal{F}$ is at level l, $\mathcal{F}(p_i) \leq l$ holds for each $p_i \in \mathcal{U}$. To safely stop the execution one should guarantee that $p_{stop} \succ p_i$ for all unread points. This is done in SaLSa by considering the *unread domain* at level l, defined as:

$$\mathcal{D}(\mathcal{F}, l) = \{p_i \in \mathcal{D} : \mathcal{F}(p_i) \leq l\}. \qquad (3)$$

Therefore, when $\mathcal{F}$ is at level l it is $\mathcal{U} \subseteq \mathcal{D}(\mathcal{F}, l)$. Note that $\mathcal{D}(\mathcal{F}, l)$, unlike $\mathcal{U}$, does not depend on the specific relation r. Then, SaLSa can safely stop fetching tuples iff the following is true:

$$\forall p_i \in \mathcal{D}(\mathcal{F}, l) : p_{stop} \succ p_i \qquad (4)$$

The subset of the $\mathcal{D}$ domain that SaLSa has not explored when it halts is denoted $\mathcal{D}(\mathcal{F}, r) \equiv \mathcal{D}(\mathcal{F}, l(\mathcal{F}, r))$.

In principle, any monotone function can be used within SaLSa. However, it is useless to use a function $\mathcal{F}$ if $\mathcal{F}$ is not able to *limit* the tuples to be read. More precisely, we say that $\mathcal{F}$ limits a relation r when $\mathcal{U}(\mathcal{F}, r) \neq \emptyset$ and that $\mathcal{F}$ is *limiting* if there exists at least a relation r for which $\mathcal{U}(\mathcal{F}, r)$ is not empty. Note that this is the same as saying that $\mathcal{D}(\mathcal{F}, r)$ is not always empty. Thus, only limiting sorting functions are worth considering for SaLSa.

3.1 Symmetric Sorting Functions

It is not clear how one could analyze the whole space of limiting functions by looking for the "best" one to use. For instance, let $d = 4$, and functions $\mathcal{F}_1 = A_1 \times (A_2 + A_3)^{A_4}$ and $\mathcal{F}_2 = (A_1 + A_2) \times \ln A_3 + A_4$. Is $\mathcal{F}_1$ better than $\mathcal{F}_2$? Are they limiting? Can we provide an effective method to check if $p_{stop} \succ \mathcal{D}(\mathcal{F}_i, l)$ when $\mathcal{F}_i$ is at level l? Can the stop point be efficiently updated? Is there any principled reason why the attributes should be assigned different roles (e.g., A_1 and A_4 in $\mathcal{F}_1$)?

Being this the first work that addresses the problem of using a sorting function to limit the number of tuples to be read, we feel it is important to focus the attention on a well-defined class of functions, which exhibit a key property of *symmetry.*

Definition 1 *A function $\mathcal{F}$ on d variables is symmetric iff it is invariant under any permutation of the variables.*

Intuitively, a symmetric function $\mathcal{F}$ does not privilege any attribute over the others, i.e., all the attributes play the same role. This seems to be a very natural requirements if one uses $\mathcal{F}$ for computing the skyline, since by definition all skyline attributes are equally important.

In the following we will only consider symmetric functions. Further, in order to avoid unnecessary complications, we will consider that all attribute values are normalized in the $[0,1]$ range, i.e., $\forall j : dom(A_j) = [0,1]$. Section 3.2 briefly touches on the case of asymmetric functions, and Section 4 describes how relations with arbitrary domains can be dealt with.

3.1.1 Choosing the Stop Point

A key factor affecting the performance of SaLSa is the choice of the stop point, p_{stop}. Clearly, a good stop point should maximize the chance of stopping earlier the execution. Although one might suspect that the choice of the stop point depends on the specific function used to sort the points, rather surprisingly this is not the case for symmetric functions. We first need the following result about symmetric monotone functions.

Lemma 1 *Let $\mathcal{F}$ be a symmetric monotone function and assume $\mathcal{F}$ is at level l. Further, let M be the maximum value such that $l = \mathcal{F}(M, 0, 0, \ldots, 0)$ and assume M is finite. Then, there is no point p for which both the following hold: 1) $\mathcal{F}(p) \leq l$, and 2) $\exists j : p[j] > M$.*

Proof. Assume that both 1) and 2) hold. Since $\mathcal{F}$ is symmetric it is possible to arbitrarily choose the attribute A_j for which $p[j] > M$. Then, let $j = 1$. Consider the point $p' = (p[1], 0, 0, \ldots, 0)$, i.e., $p'[1] = p[1]$ and $\forall j \neq 1 : p'[j] \leq p[j]$. Then, by monotonicity of $\mathcal{F}$, it is $\mathcal{F}(p') \leq \mathcal{F}(p) \leq l$. This contradicts the hypothesis that M is the maximum value for which $l = \mathcal{F}(M, 0, 0, \ldots, 0)$ holds. □

Theorem 1 *Let $\mathcal{F}$ be* any *symmetric monotone function, and let $\mathcal{S}$ be the current set of skyline points. For each point $p_i \in \mathcal{S}$, let $\underline{p_i} = \min_j\{p_i[j]\}$. The strategy that chooses the stop point according to the following* MaxiMin *rule:*

$$p_{stop} = \arg\max_{i \in \mathcal{S}}\{\underline{p_i}\} \tag{5}$$

with ties that can be arbitrarily broken, is optimal, i.e., there is no other rule that on any relation can do better than the MaxiMin *rule.*

Proof. Consider a rule that eventually chooses as stop point $p_{bad} \in \mathcal{S}$ such that $\underline{p_{bad}} < \underline{p_{stop}}$, where p_{stop} is as in Equation 5. To prove the theorem we need to show that: 1) whenever SaLSa halts due to p_{bad}, then so it does due to p_{stop}, and 2) there exists at least a relation r for which SaLSa using p_{stop} can be stopped before than if using p_{bad}.

1) Let p_i be any point in the region, denoted as $\mathcal{D}(\mathcal{F}, r)[p_{bad}]$, not explored by SaLSa when it halts by using p_{bad} as stop point. We have that $p_{bad} \succ p_i$ iff for each attribute A_j it is $p_i[j] \leq p_{bad}[j]$, with at least one inequality being strict. From Lemma 1 we have that the following does not depend on the specific attribute j:

$$\max\{p_i[j] : p_i \in \mathcal{D}(\mathcal{F}, r)[p_{bad}]\}$$

i.e., maximal attribute values in the unread domain are the same on all attributes. Denote the above with M. Then, $p_{bad} \succ p_i$ if $M \leq p_{bad}[j]$ holds for each attribute, with at least one strict inequality. There are two cases to consider here. If $\exists j : p_{bad}[j] > \underline{p_{bad}}$, then the stop condition is equivalent to require that $M \leq \underline{p_{bad}}$. Alternatively, if $\forall j : p_{bad}[j] = \underline{p_{bad}}$, then we might stop only if $M < \underline{p_{bad}}$. Since $\underline{p_{bad}} < \underline{p_{stop}}$, in both cases the result follows.

2) Immediate. □

The definition given by Equation 5 provide an efficient, $O(1)$, method for incrementally updating the stop point. Let p_{stop} be the current stop point. When a new point p is added to the skyline, the stop point either remains unchanged or it is set to p. This only depends on which among $\underline{p}$ and $\underline{p_{stop}}$ is maximum.

Example 3 *Suppose the current skyline $\mathcal{S}$ consists of points $p_1 = (.75, .4)$, $p_3 = (.3, .8)$, and $p_7 = (.05, .9)$ (the same data are used in Example 4); then, it is $\underline{p_1} = 0.4, \underline{p_3} = 0.3, \underline{p_7} = 0.05$, thus $p_{stop} = p_1$. If point $p_2 = (.55, .5)$ is added to $\mathcal{S}$, then p_2 becomes the new stop point, since it is $\underline{p_2} = 0.5 > 0.4 = \underline{p_1}$.* □

Lemma 1 also provides an effective way to determine when SaLSa can be stopped.

Theorem 2 *Let p_{stop} be the current stop point. Let $\mathcal{F}$ be any symmetric monotone function, and assume $\mathcal{F}$ is at level l. Further, let M be the maximum value such that $l = \mathcal{F}(M, 0, 0, \ldots, 0)$. If no such finite M exists, then set $M = \infty$. If $M \leq \underline{p_{stop}}$ then* SaLSa *can be stopped. Strict inequality is required only in the particular case that $\forall j : p_{stop}[j] = \underline{p_{stop}}$.*

Proof. From Lemma 1 we know that no point p such that $\mathcal{F}(p) \leq l$ has an attribute value greater than M. This is sufficient to conclude that p is dominated by p_{stop}. □

3.1.2 Some Specific Sorting Functions

Although Theorem 1 shows that the stop point can be chosen independently of the sorting function, this does not mean that all sorting functions will behave equally well. In the following we consider some relevant cases of sorting functions, and argument about their applicability. To avoid making the problem trivial to solve, we exclude from the analysis those (unrealistic) instances that include the "ideal" point $\mathbf{1} = (1, \ldots, 1)$, and consider only skylines over at least 2 dimensions, i.e., $d \geq 2$.

3.1.2.1 Sorting by Volume.

The first symmetric monotone function we consider is the one based on the "volume" of the points, as originally proposed by the authors of SFS in [7]:

$$\mathsf{vol}[0](p) = \prod_{j=1}^{d} p\,[j]$$

We will shortly explain why we denote the function as $\mathsf{vol}[0]$ rather than simply as vol. The rationale of using $\mathsf{vol}[0]$, which also justifies its name, is that if points are uniformly distributed over $\mathcal{D}$, then $\mathsf{vol}[0](p)$ is the volume of the region dominated by p. Then, fetching first points with higher volume increases the chance of early discarding many other points, thus reducing the overall number of comparisons. Since SFS is not concerned with the problem of limiting the number of points to be read, no analysis on the effectiveness of using $\mathsf{vol}[0]$ for this purpose was given in [7] and [8].

Observation 1 *The* $\mathsf{vol}[0]$ *sorting function is not limiting.*

Intuitively, $\mathsf{vol}[0]$ is not limiting since, for any level l, $\mathcal{D}(\mathcal{F}, l)$ includes points that are maximal on one or more attributes. Without loss of generality assume $d = 2$. Let $p_1 = (1, p_1[2])$ and $p_2 = (p_2[1], 1)$. For any value of the current $\mathsf{vol}[0]$ level it is possible to choose values for $p_1[2]$ and $p_2[1]$ such that both p_1 and p_2 are in $\mathcal{U}$. Then, the only point that can dominate both is the ideal point $\mathbf{1}$. The arguments can be easily generalized to an arbitrary number of attributes d by considering d points, each one maximal on a different attribute.

Technically, to make the volume function limiting one has to exclude the 0 value from the domain. To this end, consider the function $\mathsf{vol}[1](p) = \prod_{j=1}^{d}(p\,[j] + 1)$.[3] If we have, say, a 2-dimensional skyline, now there is a chance of halting SaLSa execution as soon as the level of $\mathsf{vol}[1]$ reaches a value $l < 2$. This is because $l < 2$ guarantees that no point with maximal values either on A_1 or on A_2 is still unread. Such arguments are generalized as follows.

Observation 2 *Consider the function*

$$\mathsf{vol}[m](p) = \prod_{j=1}^{d}(p\,[j] + m) \quad (m > 0).$$

Then $l(\mathsf{vol}[m], r) < (m+1)m^{d-1}$ *for any relation* r.

Proof. (sketch) Each vertex v of the hypercube $[0,1]^d$ with coordinates $(0,\ldots,0,1,0,\ldots,0)$, for which it is $\mathsf{vol}[m](v) = (m+1)m^{d-1}$, needs to be excluded from $\mathcal{U}$, otherwise the arguments used for Observation 1 would apply here. □

A detailed analysis of the effect of m on the limiting power of $\mathsf{vol}[m]$ is beyond the scope of this paper and is left as a subject for future work. However, it is instructive to have some intuition about "how much", in geometrical terms, one can expect to limit. For this, consider the point p on the main diagonal of the hypercube for which it is $\mathsf{vol}[m](p) = (m+1)m^{d-1} = m^d(m+1)/m$. For each attribute A_j it is $p\,[j] = m(\sqrt[d]{(m+1)/m} - 1)$. Thus, even for moderately large values of d, p is already quite close to the worst possible point, i.e., $(0,\ldots,0)$, yet $\mathsf{vol}[m]$ cannot discard it.

In the following we will only consider $\mathsf{vol}[1]$, and will denote it as vol for simplicity.

3.1.2.2 Sorting by Sum of Coordinates.

An obvious alternative to vol is to sum (rather than to multiply) attribute values, that is:

$$\mathsf{sum}(p) = \sum_{j=1}^{d} p\,[j]$$

Observation 3 *The* sum *sorting function is limiting and* $l(\mathsf{sum}, r) < 1$.

This immediately follows from Theorem 2. Assume that r has a point p_{stop} with $\underline{p_{stop}} = x$, $0 < x < 1$. Then, SaLSa can be halted as soon as the current level $l \leq x$, since for any point $p \in \mathcal{D}(\mathsf{sum}, x)$ it is $\forall j : p\,[j] \leq x$, with at least an inequality which is guaranteed to be strict.

3.1.2.3 Maximum Coordinate Sort.

Arguments used to show that sum is limiting can be used to show that many other functions are limiting as well. It is therefore natural to ask if there is an "optimal" sorting function, i.e., a function that on any instance r can limit r more than any other function. The answer is affirmative, and such optimal function, max, is defined as:

$$\mathsf{max}(p) = (\max_j\{p\,[j]\}, \mathsf{sum}(p))$$

Therefore, max first sorts points considering for each of them the maximum coordinate value. Then, in case of ties, a sum on the skyline attributes is used. Note that this is needed only to guarantee the monotonicity of max, and that other monotone tie-breaking rules are possible here (e.g., vol).

Clearly, max is limiting, since when $\underline{p_{stop}} = x$, $0 < x < 1$, SaLSa can be stopped as soon as it fetches a point for which it is $l \leq x$.[4]

Theorem 3 *For each relation* r *and any symmetric sorting function* $\mathcal{F}$ *different from* max *it is:* $\mathcal{D}(\mathcal{F}, r) \subset \mathcal{D}(\mathsf{max}, r)$, *thus* $\mathcal{U}(\mathcal{F}, r) \subseteq \mathcal{U}(\mathsf{max}, r)$.

Proof. (sketch) The optimality of max follows from simple geometric arguments and the observation that, due to Theorem 1, all symmetric limiting sorting function will eventually limit r using the same stop point p_{stop}, with $\underline{p_{stop}} = x$, $0 < x < 1$. This is to say that maximal attribute values in the unread domain $\mathcal{D}(\mathcal{F}, r)$ are equal to x, independently of $\mathcal{F}$. Let $\mathcal{F} \neq \mathsf{max}$. Since $\mathcal{F}$ is monotone, then $\mathcal{D}(\mathcal{F}, r) \subset [0, x]^d \equiv \mathcal{D}(\mathsf{max}, r)$. The second part of the theorem, $\mathcal{U}(\mathcal{F}, r) \subseteq \mathcal{U}(\mathsf{max}, r)$, immediately follows. The equality case has also to be considered here, since it is possible that r has no point in $\mathcal{D}(\mathsf{max}, r) \setminus \mathcal{D}(\mathcal{F}, r)$. □

We conclude this section by observing that the *limiting* and the *filtering* power of a sorting function $\mathcal{F}$, the latter

[3] SFS actually implements the *entropy* function, $E(p) = \sum_{j=1}^{d} \log(p\,[j] + 1)$, which also avoids to incur into overflow problems. In [7] it is asserted that E yields the same order as $\mathsf{vol}[0]$. This is not true, since E produces the same order of $\mathsf{vol}[1]$, which in general is different from that induced by $\mathsf{vol}[0]$.

[4] Strict inequality is required only in the particular case when $\forall j : p_{stop}[j] = x$ and duplicates are possible.

referring to the impact $\mathcal{F}$ has on the effectiveness of the filter phase in which dominance tests are executed, are not necessarily positively correlated. This is also to say that, given functions $\mathcal{F}_1$ and $\mathcal{F}_2$, it might well be the case that $\mathcal{U}(\mathcal{F}_1, r) \subset \mathcal{U}(\mathcal{F}_2, r)$, yet sorting data using $\mathcal{F}_1$ leads to execute less dominance tests than using $\mathcal{F}_2$.

Example 4 *As an example of how the different sorting functions operate, we show below the case of a 2-dimensional skyline over a relation with 8 points. The skyline consists of points p_1, p_2, p_3, and p_7 (marked with an asterisk in the tables). The stop point is point $p_2 = (.55, .5)$, with $\underline{p_2} = .5$. In the left-most table we see that sorting by* vol *(actually* vol[1]*) cannot avoid reading all the points. Indeed, after reading p_8 the level of* vol *is $1.54 = (.4 + 1) \times (.1 + 1)$. From this one derives that the maximum possible value on either A_1 or A_2 is $.54 > .5$, thus also the last point p_6 has to be read.*

Sorting by sum *allows halting the execution without reading point p_6. After reading p_8, the level of* sum *is $l = .5$, which is sufficient to conclude that all skyline points have been seen. Note that in this particular case* sum *and* vol *yield the same order.*

Finally, sorting by max *allows SaLSa to stop before reading p_5. Although p_8 is in the region dominated by the stop point, SaLSa cannot avoid reading it. Indeed, before reading p_8, the level of* max *is .55, and from this one cannot exclude that a skyline point, say $(.4, .52)$, exists.*

As to dominance tests, it can be verified that both vol *and* max *execute 11 comparisons, whereas* sum *performs only 10 tests to determine the skyline.* □

	A_1	A_2
$p_1(^*)$	.75	.4
$p_3(^*)$	.3	.8
$p_2(^*)$	.55	.5
p_4	.3	.7
$p_7(^*)$	.05	.9
p_5	.3	.35
p_8	.4	.1
p_6	.2	.25

vol

	A_1	A_2
$p_1(^*)$	.75	.4
$p_3(^*)$	.3	.8
$p_2(^*)$	.55	.5
p_4	.3	.7
$p_7(^*)$	.05	.9
p_5	.3	.35
p_8	.4	.1
p_6	.2	.25

sum

	A_1	A_2
$p_7(^*)$	.05	.9
$p_3(^*)$	.3	.8
$p_1(^*)$	.75	.4
p_4	.3	.7
$p_2(^*)$	.55	.5
p_8	.4	.1
p_5	.3	.35
p_6	.2	.25

max

3.2 Lexicographic Sort and Asymmetric Functions

In this section we try to shed some light on what the use of an asymmetric sorting function is going to change in the choice of the stop point and in the logic for halting the execution of SaLSa. We start with the well-known case of *lexicographic sort*.

A popular way to sort data is by first ordering them according to A_1 values, then breaking ties using A_2, and so on. According to this lexicographic sort, each point is assigned the (vector) value:

$$\mathsf{lex}(p) = (p\,[1], \ldots, p\,[j], \ldots, p\,[d])$$

and we write $\mathsf{lex}(p) > \mathsf{lex}(p_i)$ if p precedes p_i in the lexicographic order. Note that lexicographically sorting only on a proper subset of skyline attributes would not yield a monotone order in case of ties on the sorting attributes. This is to say that we could have $p_i \succ p$ even if p has been read before than p_i.

Since lex is not symmetric, arguments used in Section 3.1 do not apply here. In particular, the choice of the stop point cannot be based on the MaxiMin rule described in Theorem 1. Consider a point $p = (p\,[1], \ldots, p\,[k], 1, \ldots, 1)$, i.e., p is maximal on attributes $k+1, \ldots, d$. Then, p can be used to stop the execution as soon as SaLSa reads a point p_i such that $p_i[j] = 0$, $1 \leq j < k$, and $p_i[k] \leq p\,[k]$.[5] The same holds even when $k = d$, in which case one must wait to read a point $p_i = (0, \ldots, 0, p_i[d])$, $p_i[d] \leq p\,[d]$ before being able to stop. The following result summarizes such observations.

Theorem 4 *Let $\mathcal{S}$ be the current set of skyline points. For each point $p_i \in \mathcal{S}$, consider its "reversed" version, $\overleftarrow{p_i} = (p_i[d], \ldots, p_i[1])$. Then, the strategy that chooses the stop point according to the following* ReverseLex *rule:*

$$p_{stop} = \arg\max_{i \in \mathcal{S}} \{\mathsf{lex}(\overleftarrow{p_i})\} \tag{6}$$

is optimal for lex.

Rather surprisingly, above theorem says that the best point to use for stopping the execution is indeed the one that is optimal when the priority of attributes is *reversed* with respect to the one adopted for sorting.

We now turn to consider the case when $\mathcal{F}$ is a generic asymmetric sorting function. For the sake of definiteness, assume $\mathcal{F}$ is real valued, $\mathcal{F}(p) \in \Re$. The basic observation is now that when $\mathcal{F}$ is asymmetric, it is no longer true that maximal values of points in the unread domain at level l, $\mathcal{D}(\mathcal{F}, l)$, are the same for all attributes, thus results in Section 3.1 need to be generalized. This can be done as follows.

For point p_i define its j-th stop level as:

$$l_i[j] = \mathcal{F}(0, \ldots, 0, p_i[j], 0, \ldots, 0)$$

and $\underline{l_i} = \min_j\{l_i[j]\}$. Note that when the level of $\mathcal{F}$ is $l \leq \underline{l_i}$ we have that, for each j, the maximal value of A_j in $\mathcal{D}(\mathcal{F}, l)$, M_j, is guaranteed to be not higher than $p_i[j]$, $M_j \leq p_i[j]$. Then, we can generalize the MaxiMin rule of Theorem 1 by choosing as stop point:

$$p_{stop} = \arg\max_{i \in \mathcal{S}} \{\underline{l_i}\} \tag{7}$$

For instance, let $\mathcal{F} = 1.2A_1 + A_2 + 2A_3$ and consider points $p_1 = (.3, .7, .5)$ and $p_2 = (.6, .4, .3)$. We have $l_1 = (.36, .7, 1)$ and $l_2 = (.72, .4, .6)$, thus $\underline{l_1} = .36$ and $\underline{l_2} = .4$, from which we conclude that the stop point is p_2 and that we must wait $\mathcal{F}$ to reach level $l \leq .4$ before being able to stop.

4. MAKING SALSA TO WORK

In this section we briefly describe how some practical issues that arise when implementing SaLSa on top of a real DBMS can be dealt with.

In our discussion, we have assumed that all skyline attributes have the numerical domain $[0, 1]$. This was only done with the aim to simplify the presentation of results. However, in real cases each attribute might likely have a different, specific, domain. For instance, in Example 1 the domain of the Price attribute is the set of real numbers, while the domain of Rating is the set of integers from 1 to 5. Further, the domain of Location is not even numeric.

[5] The last should be a strict inequality only if $p\,[j] = 0$, $1 \leq j < k$.

For arbitrary numerical domains the problem can be easily solved as follows. Given a skyline attribute A_j, one first determines the maximum, $\max_j$, and minimum, $\min_j$, values of A_j, after that it normalizes values in the $[0,1]$ range by using in the function $\mathcal{F}$

$$\frac{A_j - \min_j}{\max_j - \min_j}$$

in place of A_j (or complementing to 1 the above if A_j has to be minimized). The specific method used for determining $\max_j$ and $\min_j$ can vary depending on the specific scenario at hand. For instance, both values might be derivable from the semantics of the application, or they could be obtained from the server catalogs and then cached in the client to be reused in other queries.

Consider now the case of categorical attributes. The preferences we consider always induce a *weak order* over attribute values. This is to say that a preference on attribute A_j implicitly defines a number n of "levels", $L_1, \ldots, L_n$, on the domain of A_j. For instance, the preference over the Location attribute in Example 1 induces the order: 'sea' $\succ$ {all other values but 'city'} $\succ$ 'city', which consists of $n = 3$ levels. Note that a value at level L_i is better than any value at level L_j, $j > i$.

Preferences of this kind cannot be directly embedded in an ORDER BY clause, since the SQL standard does not provide any means to order categorical attributes in an arbitrary way. The approach we take is to map the preference levels into n different numerical values, $x_1, \ldots, x_n \in [0,1]$, such that $L_i \succ L_j$ leads to $x_i > x_j$. A basic way for doing this is to set $x_i = (n-i)/(n-1)$, thus the first level has value 1 and the worst has value 0.

In order to provide a numerical value that can be inserted into an ORDER BY clause, one can use CASE expressions, which are part of the SQL99 standard. Putting all together, the query in Example 1 would be transformed in the standard SQL query (assuming to use sum as sorting function and that hotels' prices range from 20 to 300 dollars):

```
SELECT * FROM Hotels
ORDER BY ((300-Price)/280 + (Rating-1)/4 +
CASE Location WHEN ‘sea’ THEN 1
WHEN ‘city’ THEN 0
ELSE 0.5 END) DESC
```

Finally, the max sorting function can be easily implemented within a DBMS as a *user-defined function* (UDF). In our experimental setup we followed this approach and implemented max as a binary C UDF within IBM DB2 UDB. The computation of max over 3 or more attributes was done by nesting the function calls (e.g., $\mathsf{max}(A_1, \mathsf{max}(A_2, A_3))$).

5. EXPERIMENTAL RESULTS

We experimented with the SaLSa and the SFS algorithms using both synthetic and real datasets:

Synthetic: We used the synthetic data generator provided by the authors of [4], creating 3 different datasets (uniform, correlated, and anti-correlated), each consisting of 500,000 tuples with 6 real attributes. A dummy attribute was also added, so as to have tuples of 100 bytes. Datasets like these are commonly used to evaluate performance of skyline algorithms.

Real: We downloaded from `www.basketballreference.com` the NBA dataset. This dataset consists of 19,112 tuples containing statistics of basketball players during regular seasons in the period [1946–2005]. Multiple tuples are present for players that have switched teams during the season (one for each team + one total), thus we deleted the 'total' tuples from the set, obtaining 17,791 tuples. For each player, we only retained some of his statistics, namely number of games played during the season (*gp*), points scored (*pts*), total rebounds (*reb*), assists (*ast*), field goal made (*fgm*), and free throws made (*ftm*). Other statistics were not used since they were not available for all players, e.g., steals were not considered before 1973. The resulting tuple size is 210 bytes.

All datasets were bulk-loaded into the DB2 Universal DataBase V8.1, running on a Pentium IV 3.4 GHz PC equipped with 512 MB of main memory and a 80GB Samsung SP0812C hard drive, under the Windows XP Professional operating system. SaLSa and SFS were implemented in Java on a client machine.

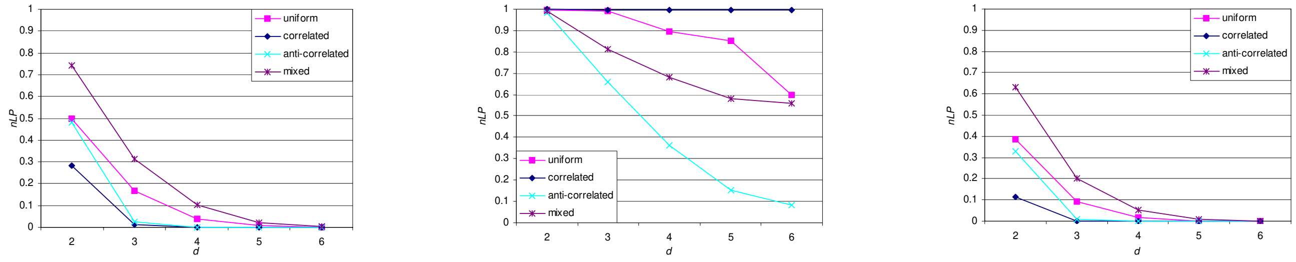
Since for all the experiments we performed the size of the skyline was sufficiently small (at most 29,295 100-bytes points) to be entirely contained in the client main memory, no temporary file was ever needed (i.e., both SaLSa and SFS always completed in a single pass).

In the first experiment we wanted to establish the performance of the vol, sum, and max sorting functions in limiting the input relation. To this end, we consider the *normalized Limiting Power* (*nLP*), which measures how many useless points are pruned when using a given sorting function. *nLP* is defined as the number of unread tuples, $|\mathcal{U}|$, divided by the number of points not in the skyline, $|r| - |\mathcal{S}|$:

$$nLP = \frac{|\mathcal{U}|}{|r| - |\mathcal{S}|} \tag{8}$$

The range of *nLP* is [0,1], with higher values meaning better limiting performance.

Values of *nLP* over the synthetic datasets are shown in Figure 1 as a function of the number of skyline attributes. Clearly, when increasing the number of attributes, finding the skyline becomes more difficult, thus *nLP* decreases. A remarkable exception is the behavior of max on the correlated dataset, in which *nLP* is almost independent on the number of attributes and always remains well above 99%. When comparing the different sorting functions, it is evident that max obtains for all distributions much higher *nLP* values than sum, which in turn always limits more than vol. The figures also include results for a *mixed* dataset, consisting of 500,000 tuples, half of which are anti-correlated and half of which are uniformly distributed in the $[0, 0.5]^d$ domain. Thus, mixed simulates the case where the "front" of the dataset is anti-correlated, after which several other tuples follow. Their actual distribution is indeed almost immaterial, the relevant thing being that such tuples are all likely to be dominated by some other tuple in the front. We found this kind of mixed distribution more similar than any of the other three synthetically generated to what real data actually look like, an example of which is given in Figure 4. This kind of distribution is the one on which both sum and vol exhibit their best limiting power, whereas max performance stays between that of uniform and anti-correlated datasets. In particular, with $d = 6$ attributes, max still prunes about 56% (≈ 270,000) tuples.



(a) (b) (c)

Figure 1: Normalized Limiting Power (nLP) for the sum (a), the max (b), and the vol (c) sorting functions as a function of the number of attributes.

While nLP measures the amount of work saved for retrieving tuples from the data server, to express the amount of work saved by the client during the computation of the skyline we introduce the *normalized Filtering Power* (nFP). We computed the number of actual dominance tests, dt, and normalized it by the maximum number of comparisons, which equals $\binom{|\mathcal{S}|}{2} + |\mathcal{S}|(|r| - |\mathcal{S}|)$. Then, nFP is just the complement to 1 of this quantity, i.e.:

$$nFP = 1 - \frac{dt}{\binom{|\mathcal{S}|}{2} + |\mathcal{S}|(|r| - |\mathcal{S}|)} \qquad (9)$$

Figure 2 shows that the max sorting function, which is the clear winner when limiting is considered, has a variable behavior in reducing the number of comparisons. In particular this is true when increasing the number of attributes, in which case the function with the best filtering power is sum, followed by vol and then by max.

Figure 3 shows the percentage of dominance tests that each function leads to execute, when compared to SFS using the same function. Thus, with respect to Figure 2, now one can appreciate the actual effect that limiting the input has on the filtering phase. Clearly, for a given function, the *relative Filtering Power*:

$$rFP = \frac{dt(\mathsf{SaLSa})}{dt(\mathsf{SFS})}$$

of SaLSa with respect to SFS is always ≤ 1, since SaLSa never executes more dominance tests than SFS. For all functions rFP tends to 1 as the number of attributes increases, with the exception of max over uniform and correlated datasets. However, the ratio of dominance tests tending to 1 does not mean that SaLSa and SFS execute the same number of comparisons. For instance, with $d = 6$ attributes and using max, SaLSa saves about 6.6×10^6 tests on the mixed and 10^7 tests on the anti-correlated datasets, respectively.

Then, we analyzed the performance of SFS and SaLSa on the NBA real dataset. For this set of experiments we considered only SFS using the vol function. The attributes were considered in the following order: *gp*, *pts*, *reb*, *ast*, *fgm*, and *ftm*. To provide a graphical intuition about the dataset, Figure 4 shows the projection of the 17,791 tuples over the *reb* and the *ast* attributes. As anticipated, it is hard to model this kind of distribution with one of the three "basic" synthetic datasets (uniform, correlated, and anti-correlated) that are commonly used to benchmark skyline algorithms. The *mixed* dataset we introduced indeed seems to provide a better fit of what one can expect in real situations.

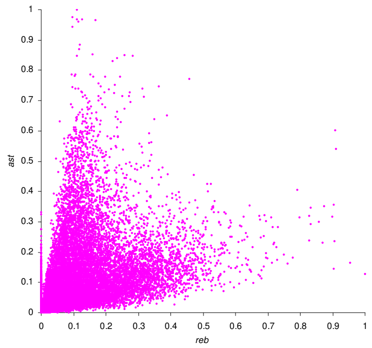


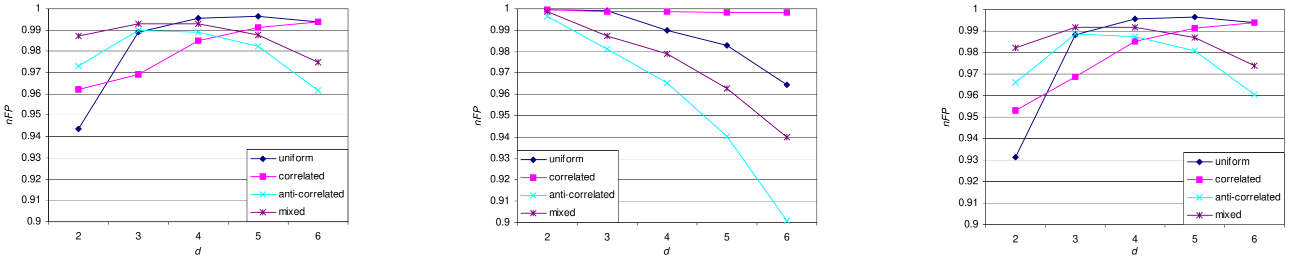
Figure 4: Visual analysis of correlation between *reb*/*ast* attributes in the NBA dataset.

Graphs in Figure 5 show, respectively, the number of fetched tuples and of comparisons for the four methods under analysis (SFS with vol, SaLSa with vol, sum, and max).

Results in Figure 5 (a) demonstrate that SaLSa is indeed highly effective in reducing the number of tuples to be fetched from the data server. For instance, even with 6 attributes SaLSa using max correctly computes the skyline without having to read about 5,500 tuples out of 17,791. Only slightly worse is the performance when either vol or sum is used, both pruning about 4,000 tuples. It is worth noting that the jump observed in the graphs of all SaLSa variants when $d = 4$ is mainly due to the insertion of the *ast* attribute, which has a very low correlation with *reb* (see Figure 4).

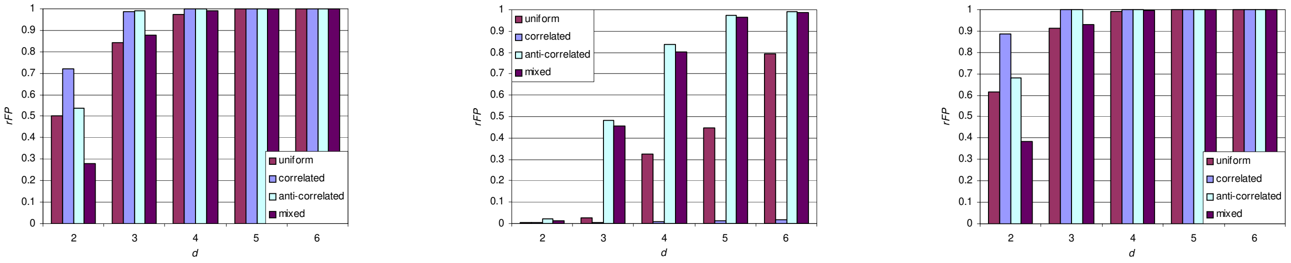
As to the number of comparisons (Figure 5 (b)), all solutions are somewhat comparable for a number of attributes $d \geq 4$, whereas for 2 and 3 attributes max is clearly the most effective method. For instance, when $d = 2$, max executes only 3,193 comparisons versus the 20,234 of SFS.

Finally, we evaluated actual computation times for the SFS and the SaLSa algorithms on the NBA dataset. In particular, we considered *sorting times*, i.e., the time needed by the data server to sort the data, *filtering times*, i.e., the time spent by the client in computing the skyline, and *communication times*, i.e., the time needed to transmit data from the server to the client. The client used in the experiments was a PC with the same HW characteristics of the data server, whereas we simulated the use of a wireless 802.11b connection with a realistic actual transfer rate of 1Mbps (see Footnote 1).



(a) (b) (c)

Figure 2: Normalized Filtering Power (nFP) for SaLSa, using sum (a), max (b), and vol (c) for sorting, as a function of the number of attributes.



(a) (b) (c)

Figure 3: Relative Filtering Power (rFP) of SaLSa wrt SFS, using sum (a), max (b), and vol (c) for sorting, as a function of the number of attributes.

Sorting and filtering costs are shown in Figure 6 (a) and (b), respectively. It can be observed that, albeit max is the best solution for both limiting and filtering points, its sorting time is the highest one. In particular, the sorting time grows linearly for max, whereas it stays almost constant when vol or sum is used. From this we conclude that when $d = 2$ we are just paying the overhead of invoking a user-defined function in DB2, whereas the linear trend only depends on our binary implementation of max, which makes such overhead linearly dependent on d.

When communication costs are considered, the higher number of tuples that SFS needs to fetch from the server has a major impact, and the limiting power becomes predominant in determining the overall performance. Thus, as Figure 6 (c) shows, SFS is the worst among the considered solutions, while the best limiting power of max wrt sum and vol makes sorting by the maximum coordinate value the cheapest alternative along the full range of considered dimensionalities.

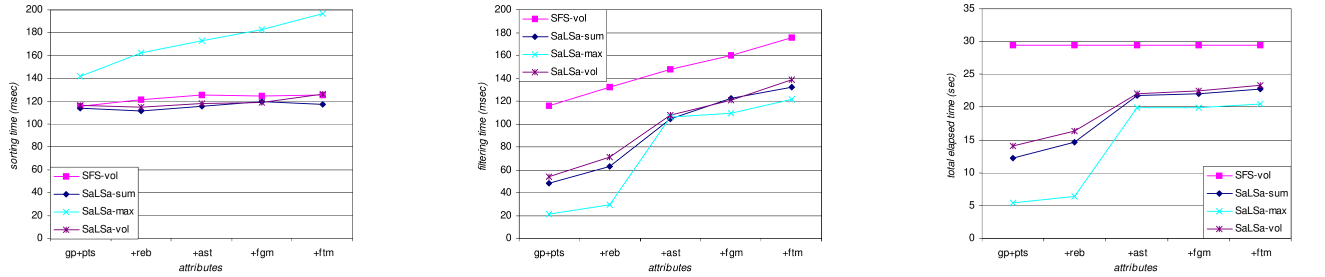
To study the impact of different transfer rates on the total elapsed time, we considered computing the skyline over 4 attributes and varied the transfer rate between the data server and the client in the range [1,100] Mbps. Figure 7 shows total computation times for the three SaLSa variants, normalized with respect to SFS costs: in the range of considered transfer rate values SaLSa always performs better than SFS (normalized costs are always less than 1), and only when communication is very fast (100 Mbps) the higher sorting costs of max outweigh the higher number of fetched tuples of sum and vol, making these two slightly better than max.

Finally, we observe that in the above experiments we have considered a client with the same computing capabilities of the server. Had we experimented with a slower client, the net effect would have been a relative increase of filtering with respect to sorting times, thus a further increase in performance of SaLSa variants with respect to SFS.

6. CONCLUSIONS

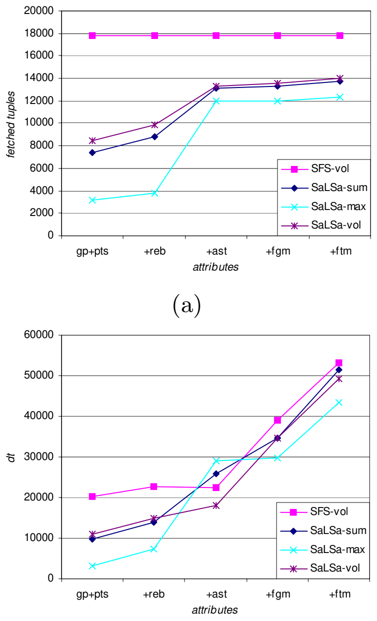
In this paper we have introduced SaLSa, a novel algorithm for computing the skyline of a relation. With respect to previous algorithms, SaLSa innovative feature is the ability of computing the correct result without having to apply dominance tests to all the objects in the relation. This is achieved by pre-sorting the data using a monotone *limiting* function, and then checking that unread data are all dominated by a so-called *stop point*. Experimental results show that this strategy is indeed effective, thus particularly attractive when one has no direct control on the data server or when the skyline logic runs on a client with limited bandwidth connection. Incidentally, the idea of limiting the amount of data to be read by exploiting the value of a monotone function is also used by the recent SUBSKY algorithm [15] for computing skylines in subspaces. However, being SUBSKY based on a fixed ordering for each attribute, it cannot be used for arbitrary preference specifications (e.g., the one on Location in Example 1).

In this paper we have considered three specific *symmetric* sorting functions, namely volume (vol), sum of coordinates (sum), and maximum coordinate (max), and proved that the latter has an optimal limiting performance. In the future, we would like to better understand the interplay between the limiting power of a sorting function and its effectiveness in reducing the number of dominance tests. A further interesting issue would be to understand how information on data distribution could be exploited to dynamically choose the best, possibly asymmetric, sorting function to use.



(a) (b) (c)

Figure 6: Elapsed times on the NBA dataset as a function of the number of attributes: sorting (a), filtering (b), and total (c).



(a)

(b)

Figure 5: Number of fetched tuples (a) and of comparisons (b) as a function of the number of attributes for the NBA dataset.

Finally, although SaLSa was conceived as a client-side algorithm, the idea of limiting the input could be applied to enhance the performance of server-side algorithms as well. Towards this direction we plan to integrate SaLSa with LESS [9], which is nowadays the best server-side algorithm when no indices are available.

7. REFERENCES

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

[2] I. Bartolini, P. Ciaccia, V. Oria, and T. Özsu. Flexible integration of multimedia sub-queries with qualitative preferences. *Multimedia Tools and Applications*. To appear.

[3] I. Bartolini, P. Ciaccia, V. Oria, and T. Özsu. Integrating the results of multimedia sub-queries using qualitative preferences. In *MIS 2004*, College Park, MD, Aug. 2004.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline operator. In *ICDE 2001*, pages 421–430, Heidelberg, Germany, Apr. 2001.

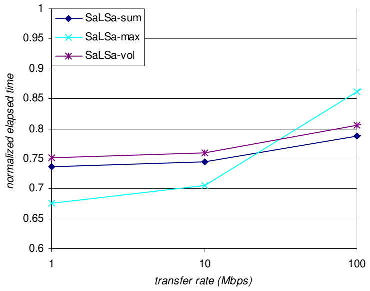


Figure 7: Elapsed time on the NBA dataset as a function of the transfer rate, normalized wrt SFS (4 attributes: *gp*, *pts*, *reb*, and *ast*).

[5] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE 2006*, Atlanta, GA, Apr. 2006.

[6] J. Chomicki. Preference formulas in relational queries. *ACM Transactions on Database Systems*, 28(4):427–466, 2003.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. Technical Report CS-2002-04, York University, Toronto, ON, Oct. 2002.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE 2003*, pages 717–719, Bangalore, India, Mar. 2003.

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

[10] W. Kießling. Foundations of preferences in database systems. In *VLDB 2002*, pages 311–322, Hong Kong, China, Aug. 2002.

[11] W. Kießling and G. Köstler. Preference SQL—Design, implementation, experiences. In *VLDB 2002*, pages 990–1001, Hong Kong, China, Aug. 2002.

[12] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB 2002*, pages 275–286, Hong Kong, China, Aug. 2002.

[13] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD 2003*, pages 467–478, San Diego, CA, June 2003.

[14] F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction*. Springer, 1985.

[15] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient computation of skylines in subspaces. In *ICDE 2006*, Atlanta, GA, Apr. 2006.