Skyline with Presorting

Jan Chomicki

Parke Godfrey

Jarek Gryz

Dongming Liang

Technical Report CS-2002-04

October 2002

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Skyline with Presorting

Jan Chomicki[1] Parke Godfrey[2,3] Jarek Gryz[3] Dongming Liang[3]

[1]University at Buffalo
201 Bell Hall, Box 602000
Buffalo, NY 14260-2000
U.S.A.
chomicki@cse.buffalo.edu

[2]College of William and Mary
P.O. Box 8795
Williamsburg, VA 23187-8795
U.S.A.
godfrey@cs.wm.edu

[3]York University
4700 Keele Street
Toronto, ON M3J 1P3
Canada
{jarek, liang}@cs.yorku.ca

Abstract

There has been interest recently in skyline queries, also called Pareto queries, on relational databases. Relational query languages do not support search for "best" tuples, beyond the order by statement. The proposed skyline operator allows one to query for best tuples with respect to any number of attributes as preferences.

The skyline operator offers a powerful mechanism for expressing preference queries over relational databases. Straightforward evaluation of skyline queries by current relational engines, however, is prohibitively expensive. An efficient algorithm for skyline is needed.

In this work, we explore what the skyline means, and why skyline queries are useful, particularly for expressing preference. We develop an algorithm for computing skyline queries that is well-behaved and efficient, particularly within a relational setting. There have been two substantial efforts to date to develop algorithms for computing skyline in the relational context. Our algorithm improves on these in efficiency, pipelinabilty of output (of the skyline tuples), stability of run-time performance, and being applicable in any context.

1 Introduction and Motivation

Often one would like to query a relational database in search of a "best" match, or tuples that best match one's preferences (in addition to matching one's necessary criteria). Relational query languages provide only limited support for this: the min and max aggregation operators, which act over a single column; and the ability to *order* tuples with respect to their attribute values. In SQL, this is done with the order by clause. This is sufficient when one's preference is synonymous with the values of one of the attributes, but is far from sufficient when one's preferences are more complex, involving more of the attributes.[1]

Consider a table of restaurant guide information, as in Figure 1. Column **S** stands for *service*, **F** for *food*, and **D** for *decor*. Each is scored from 1 to 30, with 30 as the best. This table is modeled on the Zagat Survey Guides (for example, [18]). We are interested in choosing a restaurant from the guide, and we are looking for a best choice, or best choices from which to choose. Ideally, we would like the choice to be the best for service, food, *and* decor, *and* be the lowest priced. However, there is no restaurant that is better than all others on every criterion individually, as is usually the case in real life, and in real data. No one restaurant "trumps" all others. For instance, Summer Moon is best on food, but Zakopane is best on service.

restaurant	S	F	D	price
Summer Moon	21	25	19	47.50
Zakopane	24	20	21	56.00
Brearton Grill	15	18	20	62.00
Yamanote	22	22	17	51.50
Fenton & Pickle	16	14	10	17.50
Briar Patch BBQ	14	13	3	22.50

Figure 1: Example restaurant guide table, GoodEats.

While there is no one best restaurant with respect to our criteria, we want at least to eliminate from consideration those restaurants which are worse on all criteria than some other. Thus, the Briar Patch BBQ should be eliminated because the Fenton & Pickle is better on all our criteria and is thus a better choice. The Brearton Grill is in turn eliminated

[1]To be fair, given query composability in relational query languages as SQL, much can be accomplished simply with max. However, we shall demonstrate that more support is warranted.

because Zakopane is better than it on all criteria. If Zakopane were not in the table, the Brearton Grill would remain a consideration. (Note that Summer Moon is not better than the Brearton Grill on D, decor, while it is better on every other criterion.) Meanwhile the Fenton & Pickle is worse on every criterion than every other (remaining) restaurant, except on price, where it is the best. So it stays in consideration. (If we were to remove price as one of our criteria, then the Fenton & Pickle should be eliminated too.) This would result in the choices in Figure 2.

restaurant	**S**	**F**	**D**	**price**
Summer Moon	21	25	19	47.50
Zakopane	24	20	21	56.00
Yamanote	22	22	17	51.50
Fenton & Pickle	16	14	10	17.50

Figure 2: Restaurants in the skyline.

In [4], a new relational operator is proposed which they name the *skyline operator*. They propose an extension to SQL with a skyline of clause as counterpart to this operator that would allow the easy expression of the restaurant query we imagined above. In [12, 13] and elsewhere, this is called the *Pareto operator*. Indeed, the notion of Pareto optimality with respect to multiple parameters is equivalent to that of choosing the non-dominated tuples, designated as the skyline. In [6], a more general operator called *winnow* is introduced for the purpose of expressing preference queries. Skyline is a special case of winnow.

```
select ... from ... where ...
    group by ... having ...
    skyline of a1 [min | max | diff], ...,
               an [min | max | diff]
```

Figure 3: A proposed skyline operator for SQL.

The skyline of clause is shown in Figure 3. Syntactically, it is similar to an order by clause. The columns $a_1, \ldots, a_n$ are the attributes that our preferences range over. They must be of domains that have a natural total ordering, as integers, floats, and dates. The directives min and max specify whether we prefer low or high values, respectively. The directive diff says that we are interested in retaining best choices with respect to every distinct value of that attribute. Let max be the default directive if none is stated. The skyline query in Figure 4 over the table GoodEats in Figure 1 expresses what we had in mind above for choosing "best" restaurants, and would result in the answer set in Figure 2. If the table GoodEats had a column **C** for *cuisine*, we might add C diff to the skyline of clause if we wanted the best restaurants by each cuisine group.

```
select * from GoodEats
    skyline of S max, F max, D max, price min
```

Figure 4: Skyline query to choose restaurants.

Skyline queries are not outside the expressive power of current SQL. The query in Figure 5 shows how we can write an arbitrary skyline query in present SQL. The c_i's are attributes of OurTable that we are interested to retain in our query, but are not skyline criteria. The s_i are the attributes that are our skyline criteria to be maximized, and would appear in skyline of as s_i max. (Without loss of generality, let us only consider max and not min.) The d_i are the attributes that are the skyline criteria to *differ*, and would appear in skyline of as d_i diff.

```
select c1, ..., ck, s1, ..., sm, d1, ..., dn
    from OurTable
except
select D.c1, ..., D.ck, D.s1, ..., D.sm,
       D.d1, ..., D.dn
    from OurTable T, OurTable D
    where D.s1≤ T.s1 and ... D.sm≤ T.sm and
          (D.s1< T.s1 or ... D.sm< T.sm) and
          D.d1= T.d1 and ... D.dn= T.dn
```

Figure 5: SQL for generating the skyline set.

Certainly it would be cumbersome to need to write skyline-like queries in this way. The skyline clause is a useful syntactic addition, therefore, if we encounter many queries of this nature. More important than ease of expression, however, is the expense of evaluation. The query in Figure 5 can be quite expensive. It involves a self-join over a table, and this join is a θ-join, not an equality-join. The self-join effectively computes the tuples that are trumped—or *dominated*—by other tuples. The tuples that remain, that were never trumped, are then the skyline tuples. It is known that the size of the skyline tends to be small, with certain provisos, with respect to the size of the table.[2] Thus, the intermediate result-set before the except can be enormous.

No current query optimizer would be able to do much with the query in Figure 5 to improve performance. If we want to support skyline queries, it is necessary to develop an efficient algorithm for computing skyline. And if we want the skyline operator as part of SQL, this algorithm must be easy to integrate in relational query plans, be well-behaved in a

[2] In [8], we establish that the *average-case* number of skyline tuples is $\Theta((\ln(n))^{d-1}/(d-1)!)$, in which n is the number of tuples in **R** and d is the number of dimensions of the skyline, given an independence assumption across attributes and an assumption of a sparse distribution of values.

relational context, work in all cases (without special provisions in place), and be easily accommodated by the query optimizer.

In this paper, we explore what the skyline means, and why skyline queries are useful, particularly for expressing preference. We develop a well-behaved, efficient algorithm for computing skyline queries. There have been two substantial efforts to date to develop algorithms for computing skyline in the relational context [4, 17]. Our algorithm improves on these in efficiency, pipelinabilty of output (of the skyline tuples), stability of run-time performance, and being applicable in any context.

In Section 2, we present background and discuss the related work. In Section 3, we explore the utility of skyline for preference queries, and establish the results upon which our skyline algorithm is based. In Section 4, we present the algorithm, and then optimizations on the basic approach. In Section 5, we present timing experiments and comparisons of the new algorithm with existing approaches. In Section 6, we provide insight on the issues involved in computing skyline, show points for future work, and conclude.

2 Background and Related Work

Recent years have brought new interest in expressing preference queries in the context of relational databases and the World Wide Web. Two competing approaches have emerged so far. In the first approach [1, 9], preferences are expressed by means of preference (utility) functions. The basic idea is to define a function over a relation and derive a score for each tuple. A preference query returns tuples rank-ordered according to their scores. In [1], a theoretical framework is provided for the approach, which shows how to combine preferences. In [9], how preference queries can be efficiently evaluated by using materialized views that have been precomputed and stored is demonstrated. Neither paper, however, addresses the issue of how the preference functions are derived. In the examples provided in [1], tuples have their scores already assigned. Clearly, this cannot be done manually by a user. The preference function defined in [9] offers a way of generating scores for individual tuples with minimal user input. A score of a tuple is defined as a weighted average over the attributes' values: $v_1 * a_1 + \ldots + v_n * a_n$, where $a_1, \ldots, a_n$ are the values of attributes $A_1, \ldots, A_n$ and $v_1, \ldots, v_n$ are the weights the user assigns to the attributes (that is, how much a user "cares" about the attribute). Although this approach does not require much input from a user (hence is practical), it also severely restricts the types of preferences that can be expressed: it applies only to numeric attributes. The main problem with this approach is that the score of a tuple grows monotonically and uniformly with the value of an attribute. This does not seem to correspond to realistic preferences. For instance, having ten baths in a house is not necessarily better than having five.

The second approach [4, 17] to expressing preferences is based on the `skyline` operator described in the previous section. Clearly, only a small fraction of user preferences [6] can be expressed using this operator. But the simplicity of `skyline` has its advantages as well. In particular, it is straightforward for a user to specify its parameters (that is, the attributes of interest and the ordering of their values). Secondly, skyline queries would be trivially expressible in SQL.

The emphasis of research work in this area has been in efficient implementations of the `skyline` operator. In [4], several algorithms for computing skyline queries are presented. The basic algorithm, a block-nested loops (or BNL for short) repeatedly scans a set of tuples. For each iteration, a window of incomparable tuples is kept in memory. When a tuple from the input relation is compared with the tuples in the window it may: (a) be dominated by a tuple in the window, in which case it is discarded; (b) be incomparable to any tuples in the window, in which case it is added to the window; or (c) dominate some tuples in the window, in which case it is added to the window and the dominated tuples are discarded. Multiple iterations are necessary if the window is not big enough to store all of the generated incomparable tuples. A tuple in the window becomes a part of the skyline once it has been compared to the rest of the tuples (modulo eliminations) and survived.

In [4], another algorithm, divide-and-conquer, is considered which in some cases provides better performance than BNL. However, in all experiments presented in [4] and [17], BNL performs better for small skylines and up to five dimensions and is uniformly better in terms of I/O. The divide-and-conquer algorithm would not scale well for larger datasets—or smaller buffer pools—than used in [4]. For this reason, we only discuss BNL algorithm in this paper.

In [4] and in [17], both consider indexes for more efficient computation of skyline queries. However, indexes suffer from many limitations in this context. There are two extreme solutions to index design for skyline queries. One way is to build an index storing the values of attributes relevant for skyline. Most of the computational effort in skyline computation, unfortunately, is not on efficient access to the data, but on its manipulation (that is, skyline computation is CPU-bound). Thus, it may seem that a better solution for an index design is to precompute as

much as possible (or all) of the skyline. Such an index, however, is fragile in the face of updates: a single insertion of a tuple that dominates the current skyline would invalidate the entire index. Additionally, unless the index explicitly stores the values of the skyline attributes, it cannot be used to compute a skyline of any subset of these attributes. Similarly, a skyline of a set of attributes cannot be computed from skylines of the subsets of its attributes. This is because, in general, (skyline of $a_1, \ldots, a_k \cup$ skyline of $a_{k+1}, \ldots, a_n$) $\subsetneq$ skyline of $a_1, \ldots, a_n$. In [14], the use of spatial indexes is considered to achieve pipelinability of the results. However, this algorithm for the skyline operator is not composable with other relational operations such as selection, so it of limited utility.

The most important argument against using indexes to compute skyline queries is the fact that the skyline operator is *holistic*, in the sense of holistic aggregation operators. This implies that the skyline operator is not, in general, commutative with selections.[3] For any skyline query with a select condition, most likely the index cannot be used.

Skyline computation is similar to the *maximal vector problem* studied in [2, 15, 16]. These consider algorithmic solutions to the problem and address the issue of skyline size. None of these works addresses the problem in a database context, however. In [7], we address the question of skyline query cardinality more concretely.

The closest related area in databases is the nearest neighbor problem [3, 10, 11]. Although the nearest neighbor problem is different from skyline computation, we believe that it can provide an alternative approach when the size of the skyline is unmanageable.

3 Skyline versus Ranking

The skyline of a relation in essence represents the best tuples of the relation, the Pareto optimal "solutions", with respect to the skyline criteria. Another way to find "best" tuples is to score each tuple with respect to one's criteria (call these preferences), and then choose those tuples with the best score (ranking). The latter could be done efficiently in a relational setting. In one table scan, one can score the tuples *and* collect the best scoring tuples.[4]

In what ways is skyline interesting then, and how is it related to ranking? It is known that the skyline represents the closure over the maximum scoring tuples of the relation with respect to all *monotone scoring functions*. For example, in choosing a restaurant as in the example in Section 1, say that one values service quality twice as much as food quality, and food quality twice as much as decor, those restaurants that are best with respect to this "weighting" will appear in the skyline. Furthermore, the skyline is the least-upper-bound closure over the (maximums of) the monotone scoring functions [4].

This means that the skyline can be used instead of ranking, or it can be used in conjunction with ranking. First, since the best tuples with respect to any (monotone) scoring are in the skyline, one only needs effectively to query the skyline with one's preference queries, and not the original table itself. The skyline is (usually) significantly smaller than the table itself [7], so this would be much more efficient if one had many preference queries to try over the same dataset. Second, as defining one's preferences in a preference query can be quite difficult, while expressing a skyline query is relatively easy, users may find skyline queries beneficial. The skyline overanswers with respect to the users' intent in a way, since it includes the best tuples with respect to *any* preferences. So there will be some choices (tuples) among the skyline that are not of interest to the user. However, every best choice with respect to the user's implicit preferences shows up too. Lastly, there are interesting choices which have strong intuitive appeal that can show up in the skyline, but which are exceedingly difficult to find by ranking.

While in [4], they observe this relation of skyline with monotone scoring functions, they did not offer proof nor did they discuss *linear scoring functions*, to which much work restricts focus. Let us investigate this more closely, and more formally, then, for the following reasons:

- to relate skyline to preference queries, and to illustrate that expressing preferences by scoring is more difficult than one might initially expect;
- to rectify some common misconceptions regarding scoring for the purposes of preference queries, and regarding the claim for skyline; and
- to demonstrate a useful property of monotone scoring that we can exploit for an efficient algorithm to compute the skyline (Section 4).

Let attributes $a_1, \ldots, a_k$ of schema $\mathcal{R}$ be the skyline criteria, without loss of generality, with respect to "max".[5] Let the domains of the a_i's be real, without loss of generality. Let **R** be a relation of schema $\mathcal{R}$, and so represents a given instance.

Definition 1 *Define a* monotone scoring function S *with respect to* $\mathcal{R}$ *as a function that takes as its input*

[3] In [6], cases of commutativity of skyline with other relational operators are shown.

[4] The best scoring tuples can be collected in this single pass as long as each score group fits in main memory.

[5] We also ignore "diff" in this discussion, without loss of generality. The diff in essence allows skyline to be combined with group by: for each group of diff values, the skyline is computed. The issue of diff is important for the design of an algorithm for computing skyline, however. A general algorithm for skyline must be able to accommodate diff efficiently.

domain tuples of $\mathcal{R}$, and maps them onto the range of reals. S is composed of k monotone increasing functions, $f_1, \ldots, f_k$. For any tuple $t \in \mathbf{R}$,

$$S(t) = \sum_{i=1}^{k} f_i(t[\mathsf{a}_i])$$

Lemma 2 *Any tuple that has the best score over $\mathbf{R}$ with respect to any monotone scoring function S with respect to $\mathcal{R}$ must be in the skyline.*

Proof. Assume that this is not the case. Let $t, r \in \mathbf{R}$, $S(t)$ be a maximum score, and $S(t) > S(r)$, but $t[\mathsf{a}_i] \leq r[\mathsf{a}_i]$, for $i \in 1, \ldots, k$. Clearly then,

$$\sum_{i=1}^{k} f_i(t[\mathsf{a}_i]) \leq \sum_{i=1}^{k} f_i(r[\mathsf{a}_i])$$

which means $S(t) \leq S(r)$. Contradiction. □

It is more difficult to show that every tuple of the skyline is the best score of some monotone scoring. Most restrict attention to linear weightings when considering scoring, though, so let us consider this first.

Definition 3 *Define a* positive, linear scoring function*, W, as any function over a table $\mathbf{R}$'s tuples of the form*

$$W(t) = \sum_{i=1}^{k} w_i t[\mathsf{a}_i]$$

in which the w_i's are positive, real constants.

As we insist that the w_i's are positive, the class of the positive, linear scoring functions is a proper subclass of the monotone scoring functions. Commonly in preference query work, as in [9], the focus is restricted to linear scoring. It is not true, however, that every skyline tuple is the best with respect to some positive, linear scoring.

Theorem 4 *It is possible for a skyline tuple to exist on $\mathbf{R}$ such that, for every positive, linear scoring function, the tuple does not have the maximum score with respect to the function over table $\mathbf{R}$.*

Proof. Consider $\mathbf{R} = \{\langle 4,1\rangle, \langle 2,2\rangle, \langle 1,4\rangle\}$. (The schema is $\langle \mathsf{a}_1, \mathsf{a}_2\rangle$.) All three tuples are in the skyline (skyline of a_1, a_2). Linear scorings that choose $\langle 4,1\rangle$ and $\langle 1,4\rangle$ are obvious. Let us attempt to find a linear scoring that chooses $\langle 2,2\rangle$. We need that

1. $2w_1 + 2w_2 \geq 4w_1 + 1w_2$, and
2. $2w_1 + 2w_2 \geq 1w_1 + 4w_2$.

By 1, $w_2 \geq 2w_1$. By 2, $w_1 \geq 2w_2$. Therefore, $w_1 \geq 2w_1$. There is no solution with $w_1 > 0$, and so there is no positive, linear scoring that scores $\langle 2,2\rangle$ best. □

Note that $\langle 2,2\rangle$ (in the proof above) is an interesting choice. Tuples $\langle 4,1\rangle$ and $\langle 1,4\rangle$ represent in a way outliers. They make the skyline because each has an attribute with an extrema value. Whereas $\langle 2,2\rangle$ represents a balance between the attributes (and hence, preferences). For example, if we are conducting a house hunt, a_1 may represent the number of bathrooms, and a_2, the number of bedrooms. Neither a house with four bathrooms and one bedroom, nor one with one bathroom and four bedrooms, seem very appealing, whereas a 2bth/2bdrm house might.

Theorem 5 *The skyline contains all, and only, tuples yielding maximum values of monotone scoring functions.*

Proof. One direction has been proven already by Lemma 2: any tuple that scores maximally with respect to a monotone scoring function is in the skyline. We must now prove the other direction: that, for every tuple in the skyline, there exists a monotone scoring function that scores it maximally.

We can show this as it is always possible to construct a monotone scoring S_t for any given tuple $t \in \mathbf{R}$ which is in the skyline such that $S_t(t)$ is maximum. Let the attribute values of the a_i's range over the reals from 0 to 1, non-inclusive, without loss of generality. Let S_t be composed of monotone functions as follows:

$$f_i(r) = \begin{cases} r[\mathsf{a}_i] & \text{if } r[\mathsf{a}_i] < t[\mathsf{a}_i] \\ k + r[\mathsf{a}_i] & \text{otherwise} \end{cases}$$

Thus, $S_t(t) = k^2 + \epsilon$, for some $\epsilon > 0$. Consider any tuple $r \in \mathbf{R}$ which is not equivalent to t over $\mathsf{a}_1, \ldots, \mathsf{a}_k$. Since t is in the skyline, there is an a_i such that $r[\mathsf{a}_i] < t[\mathsf{a}_i]$. Note that for any tuple q and any f_j, $f_j(q) < k + 1$. For r and f_i, we know that $f_i(r) < 1$. Therefore, $S_t(r) < (k+1)(k-1)+1 = k^2$. So, $S_t(r) < S_t(t)$. Thus, $S_t(t)$ has been shown to be the maximum. □

While there exists a monotone scoring function that chooses—assigns the highest score to—any given skyline tuple, it does not mean anyone would ever find this function. In particular, this is because, in many cases, any such function is a contrivance based upon that skyline's values, just as we contrived above. The user is searching for "best" tuples and has not seen them yet. Thus, it is unlikely anyone would discover a tuple like $\langle 2,2\rangle$ above with any preference query. Yet, the 2bth/2bdrm house might be exactly what we wanted.

We should note that not every preference can be cast via a monotone scoring [6]. Hence, skyline is not adequate for those. We restrict our attention in this paper, however, to skyline.

For the algorithm for skyline computation we are to develop, we can exploit our observations on the monotone scoring functions. Let us define the *dominance relation*, "$\preceq$", as follows: for tuples any $r, t \in \mathbf{R}$, $r \preceq t$ *iff* $r[\mathsf{a}_i] \leq t[\mathsf{a}_i]$, for all $i \in 1, \ldots, k$.

Further define that $r \prec t$ *iff* $r \preceq t$ *and* $r[\mathsf{a}_i] < t[\mathsf{a}_i]$, for some $i \in 1, \ldots, k$.

Theorem 6 *Any total order of the tuples of* **R** *with respect to* any *monotone scoring function (ordered from highest to lowest score) is a topological sort with respect to the skyline dominance partial relation ("$\preceq$").*

Proof. Assume otherwise. Thus there exist tuples $r, t \in \mathbf{T}$ and a monotone scoring function S such that $t \preceq r$ but $S(t) > S(r)$. As noted in the proof of Lemma 2, because $t \preceq r$, $f_i(t[\mathsf{a}_i]) \leq f_i(r[\mathsf{a}_i])$, for $i \in 1, \ldots, k$. However, then $S(t) \leq S(r)$, by S's definition. Contradiction. □

So if $S(r) < S(t)$, it is possible that $r \prec t$, but we are certain that $t \not\prec r$. If $S(r) = S(t)$, then either $r \preceq t$ and $t \preceq r$ (that is, they are equivalent with respect to their projection over the skyline attributes), or r and t are incomparable with respect to dominance. A skyline tuple $t \in \mathbf{R}$ is a tuple such that there is no $r \in \mathbf{R}$ such that $t \prec r$.

Consider the total ordering on **R** provided by the basic SQL order by as in the query in Figure 6. This total order is a topological sort with respect to dominance.

```
select * from R
    order by a1 desc, ..., ak desc;
```

Figure 6: An order by query that produces a total monotone order.

While the following proposition is fairly obvious, we state it and prove it for sake of insight, and because we shall exploit this proposition to build a better skyline algorithm.

Theorem 7 *Any nested sort of* **R** *over the skyline attributes (sorting in descending order on each), as in the query in Figure 6, is a topological sort with respect to the dominance partial order.*

Proof. Assume again, without loss of generality, that the domains of the a_i's are reals over 0 to 1, non-inclusive. Consider the nested sort by the query in Figure 6, without loss of generality. As **R** is finite, there is an $\epsilon > 0$ such that, for all $i \in 1, \ldots, k$, for all $r, t \in \mathbf{R}$, if $r[\mathsf{a}_i] < t[\mathsf{a}_i]$, then $t[\mathsf{a}_i] - r[\mathsf{a}_i] > \epsilon$. Consider the following scoring function:

$$T(r) = \sum_{i=1}^{k} (\epsilon/k)^{i-1} r[\mathsf{a}_i]$$

Note that if $r[\mathsf{a}_1] > t[\mathsf{a}_1]$, then $T(r) > T(t)$. More generally, the scoring function T gives total preference to a_i over all of $\mathsf{a}_{i+1}, \ldots, \mathsf{a}_k$. Any increase in the value of a_i, however small, is worth more than any cumulative increases in $\mathsf{a}_{i+1}, \ldots, \mathsf{a}_k$, however large. This ordering, then, is equivalent to the nested sort obtained by the SQL query in Figure 6. □

As we read the tuples output by the query in Figure 6 one by one, it is only possible that the current tuple is dominated by one of the tuples that came before it (if, in fact, it is dominated). It is impossible that the current tuple is dominated by any tuple to follow it in the stream. Thus, the very first tuple must belong to the skyline; no tuple precedes it. The second tuple might be dominated, but only by first tuple, if at all. And so forth.

4 The Sort-Filter-Skyline Algorithm

4.1 SFS

The observation in the last section provides us the basis for an algorithm to compute skyline. First, we sort our table as with the query in Figure 6. In a relational engine, an external sort routine can be called for this. Buffer pool space is then allocated as a *window* in which skyline tuples are to be placed as found. A cursor pass over the sorted tuples is then commenced. The current tuple is checked against the tuples cached in the window. If the current tuple is dominated by any of the window tuples, it is safe to discard it. It cannot be a skyline tuple. (We have established that the current tuple cannot dominate any of the tuples in the window.) Otherwise, the current tuple is incomparable with each of the window tuples. Thus, it is a skyline tuple itself. Note that it was sufficient that we compared the current tuple with just the window tuples, and not all tuples that preceded it. This is because if any preceding tuples were discarded, it can only be because another tuple already in the window dominated it. Since dominance is transitive, then comparing against the window tuples is sufficient. In the case that the current tuple was not dominated, if there is space left in the window, it is added to the window. Note that we can also place the tuple on the output stream simultaneously, as we know that it is skyline. The algorithm fetches the next tuple from the stream and repeats.

It can happen that the current tuple is seen to be skyline, but there is no space remaining in the window to add it. In this event, the algorithm switches modes. This tuple is written instead to a temporary file. (That is, it is written to a heapfile managed by the RDBMS.) Subsequent tuples are checked against the window as before. If a tuple is dominated, it is discarded as before. Else, if the tuple is not dominated, it is written to the temporary file instead. We can no longer be certain that such a tuple is skyline. It has been compared against the skyline tuples in the window (and been found not to be dominated), but it has not been compared against other tuples now in the temporary file, tuples which preceded it.

If the algorithm exhausts the tuple stream and no

```
unfinished = TRUE
while (unfinished)
    T = open_cursor(Heap)
    unfinished = FALSE
    while (next_tuple(T, t))
        if ("t is not dominated")
            if ("window is full")
                unfinished = TRUE
                break
            else
                "Add t to window."
    if (unfinished)
        S = open_new_file(SecondPass)
        write(S, t)
        while (next_tuple(T, t))
            if ("t is not dominated")
                write(S, t)
    free(Heap)
    close(S)
    Heap = SecondPass
    "Write window tuples to output."
    "Clear window."
```

Figure 7: The Sort-Filter-Skyline algorithm.

tuple was written to the temporary file, we are finished. All the skyline tuples have been found. Otherwise, the algorithm repeats another pass, opening the temporary file as the input stream. Eventually, on some pass, no temporary file will be written, and the algorithm will halt. Figure 7 presents the algorithm in pseudo-code. We call it Sort-Filter-Skyline (SFS). The input, Heap, is assumed to be sorted already.

If the window is large enough to hold all the skyline tuples, SFS halts in a single pass. Otherwise, multiple passes are necessary. It is important to note, however, that SFS's passes (and likewise, BNL's passes) are not passes in the traditional sense, as with external sorting. Not every pass is the over the entire table, with the same number of I/O's each time. Many tuples will be discarded during a pass, and thus the next pass will be over fewer pages. More important is to count the additional pages (thus I/O's) required by the algorithm.

4.2 SFS and BNL

SFS is quite similar to BNL from [4]. The BNL algorithm keeps a window, but new tuples may dominate window tuples which requires replacement. SFS requires initial sorting of the data, which BNL does not. In this and the next section, we shall demonstrate that SFS is the better algorithm, especially in a relational setting.

The two algorithms have relative merits. Advantages of BNL are that no preliminary sort is necessary, its input stream can be pipelined, it tends to take the minimum number of passes, and it has a naturally good *reduction factor* (which we discuss below). Advantages of SFS are that it *always* takes the minimum number of passes, its output stream (the skyline tuples) is pipelinable, it is much less CPU-bound (and thus, it is stable with respect to increased window sizes), and it can generate "interesting orders" for other relational operators. Meanwhile BNL's disadvantages are that it is quite CPU-bound (times can go *up* with larger window sizes), "bad" input ordering leads to pathological run-times, and it blocks on output. SFS's disadvantages are that a preliminary sort is required, and it consequently blocks on input.

It is really BNL's disadvantages that make it unsuitable as an algorithm in a relational engine. Skyline computation is CPU-intensive for a relational operation. CPU time is not dominated by I/O-time, thus we must account for it. In timing results in [4] and in [17], BNL exhibited to be CPU-bound, and we observed this as well. Window operations are expensive for BNL. Each new tuple must be checked against the window tuples for dominance. (SFS pays this price too.) Tuples must be discarded from the window when a new tuple dominates them. (SFS does not have this expense.) And extra bookkeeping is needed to ascertain when a window tuple is deemed skyline. (SFS does not have this expense either.)

The CPU-boundedness of BNL can be seen in that the time can actually increase for BNL when it is allocated a larger window. This is bad for a relational engine. It is assumed by the optimizer that an algorithm's performance will be improved by more buffer allocation (or at least not harmed). Devising a good cost model for BNL to integrate into the optimizer would be complex, if not impossible.

BNL can exhibit extremely bad run-times when the input tuples are ordered in a "bad" way. We demonstrate this in the next section. (It is perhaps surprising that BNL runs reasonably well on "random" ordered input.) Unfortunately, such "average-case" is not sufficient in a relational setting. If a table has a clustered (tree) index, which is quite likely, its tuples are ordered in the heapfile. Data may be ordered due to other operators before it arrives as input at a skyline operation. It is impossible to ensure that the skyline operation receives its input in a "random" ordering. This extra unpredictability in BNL's runtime, and of course, the fact that it can exhibit pathological run-times, make it only harder to accommodate in the optimizer. We shall show that SFS addresses these deficiencies while providing new benefits.

Meanwhile, we learned that BNL has some surprisingly good qualities. For random cases, it tends

to take the minimum number of passes needed, even though in theory it may require more. As discussed above, more important is that BNL in average case is very good at eliminating tuples, thus reducing the number of pages written per subsequent pass. Let us call this the algorithms' *reduction factor*. In fact, the total number of pages over *all* passes after the first usually is a small fraction of the number of pages of the initial pass (which is a complete table scan itself). BNL's reduction factor is superior to SFS's in SFS's basic version. Thus, SFS spends more I/O's. (We note that SFS is still competitive in run-time with BNL even at this point, mostly because BNL is CPU-bound.)

In the next sub-section, we introduce some optimizations for SFS. We explain why BNL's reduction factor is better than basic SFS's, and present a remedy that improves SFS's reduction factor beyond BNL's. This version of SFS performs better time-wise than BNL, and it consumes fewer I/O's. (This does not count that SFS must sort first. However, we shall show that SFS's performance is better than BNL's despite the cost of SFS's presorting.)

4.3 Optimizations

Reduction Factor

A key to efficiency for any skyline algorithm is to eliminate tuples that are not skyline as quickly as possible. In the ideal, every eliminated tuple would only be involved in a single comparison, which shows it to be dominated. In the worst case, a tuple that is eventually eliminated is compared against every other tuple with which it is incomparable (with respect to dominance) before it is finally compared against a tuple that dominates it. In cases that SFS or BNL are destined to make multiple passes, how large the run of the second pass will be depends on how efficient the window was during the first pass at eliminating tuples.

For SFS, at least, only skyline tuples are kept in the window. One might think on first glance that any skyline tuple ought to be good at eliminating other tuples, that it will likely dominate many others in the table. This is not necessarily true, however. Recall the definition of a skyline tuple: a tuple that is *not* dominated by any other. So while some skyline tuples are great dominators, no doubt, there are possibly others that dominate *no* other tuples.

Let us formalize this some, for sake of discussion. Define a function over the domain of tuples in $\mathbf{R}$ with respect to $\mathbf{R}$ called the *dominance number*, *dn*. This function maps a tuple to the number of tuples in $\mathbf{R}$ that it properly dominates ("$\prec$"). So, given that $\mathbf{R}$ has n tuples, $dn(t)$ can range from 0 to $n-1$. If tuple t is in the window (for either SFS or BNL) for the complete first pass, at least $dn(t)$ tuples will be eliminated. Of course, *dn*'s are not additive: window tuples will dominate some of the same tuples in common. However, this provides us with a good heuristic: We want to maximize the cumulative *dn* of the tuples in the window. This will tend to maximize the algorithm's reduction factor.

So why does BNL exhibit such a good reduction factor? After all, for BNL, the window holds a mix of skyline and non-skyline tuples at any given point. Note that a non-skyline tuple can have a high *dn*, however. The only thing that we can say is that, for any non-skyline tuple t, there *exists* a skyline tuple s such that $dn(s) > dn(t)$. Since dominance is transitive, s properly dominates any tuple that t does, and (properly) dominates t too. BNL replaces tuples in the window as it finds new tuples that dominate existing window tuples. Every time we replace window tuples by a new tuple, note that the new tuple's *dn* must be greater than the maximum *dn* over the tuples that it is replacing. This means that the cumulative *dn* of the window for BNL is always increasing! This results in a great reduction factor for BNL.

SFS does not have this advantage, since we purposefully designed it to eliminate the need for replacement in the window. Once the window is filled on a pass for SFS, the cumulative *dn* is fixed for the rest of the pass. Our only available strategy is to fill the window initially with tuples with high *dn*'s. This is completely dependent upon the sort order of the tuples established before we commence the filtering passes. Let us analyze what happens currently. We employ a sort as with the query in Figure 6, a nested sort over the skyline attributes. The very first tuple t_1 (which must be skyline) has the maximum a_1 value with respect to $\mathbf{R}$. Say that $t_1[\mathsf{a}_1] = 100$. Then $t_1[\mathsf{a}_2]$ is the maximum with respect to all tuples in $\mathbf{R}$ that have $\mathsf{a}_1 = 100$. This is probably high. And so forth for a_3, ..., a_k. Thus, t_1 with high probability has a high *dn*. Now consider t_i such that $t_i[\mathsf{a}_1] = 100$, but $t_{i+1}[\mathsf{a}_1] < 100$. So t_i is the last of the "$\mathsf{a}_1 = 100$" group. Its a_2 value is the *lowest* of the "$\mathsf{a}_1 = 100$" group, and so forth. With high probability, t_i's *dn* is low. However, if t_i is skyline (and it well could be), it is added to the window.

So SFS using a nested sort for its input tends to flood the window with skyline tuples with low *dn*'s, on average, which is the opposite of what we want. In Section 3, we observed that we can use *any* monotone scoring function for sorting as input to SFS. It might be tempting, if we could know tuples' *dn*'s, to sort on *dn*. The *dn* function is, of course, monotone with respect to dominance. However, it would be prohibitively expensive to calculate tuples' *dn*'s. Next best then would be to approximate the *dn*'s, which we can do.

Instead of a tuple's *dn*, we can estimate the probability that a given tuple dominates an arbitrary tuple. For this, we need a model of our data. Let us make the following assumptions. First, each skyline attribute's domain is over the reals between 0 and 1, non-inclusive. Second, the values of an attribute in **R** are uniformally distributed. Lastly, the values of the skyline attributes over the tuples of **R** are pair-wise independent. So given a tuple t and a randomly chosen $r \in \mathbf{R}$, what is the probability that $t[\mathsf{a}_i] > r[\mathsf{a}_i]$? It is the value $t[\mathsf{a}_i]$ itself, due to our uniform distribution assumption (and due to that $t[\mathsf{a}_i]$ is normalized between 0 and 1). Then the probability that $r \prec t$, given t is

$$\prod_{i=1}^{k} t[\mathsf{a}_i]$$

by our independence assumption. We can compute this for each tuple just from the tuple itself. Is this probability a monotone scoring function? It is easy to show that it is monotone. However, it is not formally a monotone scoring function as we defined this in Section 3; the definition only allowed addition of the monotone functions applied over the skyline attributes. Define the monotone scoring function E then as

$$E(t) = \sum_{i=1}^{k} \ln(t[\mathsf{a}_i] + 1)$$

This clearly results in the same order as ordering by the probability. Interestingly, this is an *entropy* measure, so let us call this monotone scoring function E *entropy scoring*.

Our first assumption can always be met by normalizing the data. Relational systems usually keep statistics on tables, so it should be possible to do this without accessing the data. The second assumption of uniform distribution of values is often wrong. However, we are not interested in the actual dominance probability of tuples, but in a relative ordering with respect to that probability. Other distributions would not effect this relative ordering much, so E would remain a good ordering heuristic in these cases. The last assumption of independence too is likely to be wrong. Even in cases where independence is badly violated, E should remain a good heuristic, as again, the relative ordering would not be greatly effected. Regardless of the assumptions, E is always a monotone scoring function over **R**, and we can always safely use it with SFS.

Projection

For SFS, a tuple is added to the window only if it is in the skyline. Therefore, the tuple at the same time can be pushed to the output. So it is not necessary to keep the actual tuple in the window. All we need is that we can check subsequent tuples for whether they are dominated by this tuple. For this, we only need the tuple's skyline attributes. Real data will have many attributes in addition to the attributes we are using as skyline criteria. Also, attributes suitable as skyline conditions are comparables, as integer, float, and date. These tend to be small, storage-wise. A tuple's other attributes will likely include character data and be relatively large, storage-wise. So projecting out the non-skyline attributes of tuples when we add them to the window can be a great benefit. Significantly more skyline tuples will fit into the same size window. Likewise, there is no need to ever keep duplicate (projected) tuples in the window. So we can do duplicate elimination, which also makes better use of the window space.

BNL cannot use this optimization, because when a tuple is added to the window, it is not known whether the tuple is in the skyline. Eventually, when it is determined that a window tuple is in the skyline, the tuple then is pushed to the output. Of course BNL must have the whole tuple available at that time.[6]

Dimensional Reduction

Another optimization available to SFS is due again to the fact that we first sort the table. Recall the nested sort that results from the query in Figure 6. Now consider the table that results from the query in Figure 8. It has precisely the same skyline as table **R**. We choose the maximum a_k for each "$\mathsf{a}_1, \ldots, \mathsf{a}_{k-1}$" group. Clearly, any tuple in the group but with a non-maximum a_k cannot belong to the skyline. Of course, we can only apply this reduction once. (Implemented internally, other attributes of **R** besides the a_i's could be preserved during the "group by" computation.)

```
select a1, ..., ak-1, max(ak) as ak from R
    group by a1, ..., ak-1;
    order by a1 desc, ..., ak-1 desc;
```

Figure 8: An order by query that produces a total monotone order.

This optimization is useful in cases when the number of distinct values for each of the attributes a_1, ..., a_{k-1} appearing in **R** is small, so that the number of groups is much smaller than the number of tuples. If one attribute has many distinct values, we

[6] BNL could be altered so that it keeps projected tuples in the window. The projection would also need to include the *record identifier*. Once the record were verified as skyline, the full record could be retrieved by the identifier. This would be quite expensive, though, if there are many skyline tuples.

can make this one our "a_k". In such a case, we are applying SFS to the result of the query in Figure 8, which can be a much smaller input table.

Diff

SFS can handle skyline of with the diff directive without problem. In fact, the presence of diff attributes in the query help the efficiency of the algorithm. In the sort phase, we would do a nested sort on the diff attributes outer-most and on the scoring function (say E), or the max / min attributes, innermost. During the filter phase, every time the diff group changes, we can can clear the window.

Note that BNL's performance is not helped by diff attributes at all. One could argue that in the query plan, the data should be sorted first on the diff attributes. BNL could be tailored to process the diff groups as does SFS. However, if the data must be sorted initially, we would choose SFS over BNL anyway.

4.4 Other Advantages of SFS

Because the output of SFS is pipelinable, this means the algorithm can be stopped early. This is useful if the user only wants some answers, or the top N answers. Research has been done on relational systems to show how *stopping early* can be accommodated, and how this can be used for optimization [5]. SFS can be used in conjunction with this.

SFS can be combined with any preference ordering. If the user has additional preferences beyond the skyline criteria—as long as these preferences are equivalent to a monotone scoring—SFS can produce the skyline in that preference order. We can use the monotone scoring function induced from the preferences for SFS's preliminary sort. This is especially useful if the user is also interested in just a few answers.

The SFS algorithm can also be adapted to find multiple *strata* of skyline. (Skyline strata are introduced in [6], in which they are called *iterations*.) Let $sky(\mathbf{R})$ denote the skyline of $\mathbf{R}$. Let us define stratum $s_i(\mathbf{R})$ as follows.

$$s_i(\mathbf{R}) = sky(\mathbf{R} - \bigcup_{j=0}^{i-1} s_j(\mathbf{R}))$$

Thus, $s_0(\mathbf{R}) = sky(\mathbf{R})$. Stratum $s_1(\mathbf{R})$ is the skyline of $\mathbf{R}$ with the initial skyline tuples removed. And so forth. Strata can be useful for users querying for choices. Consider our restaurant search from Section 1, and assume that there is a restaurant perfect on all accounts: scores of 30's for food, service, and decor, and it is free. The skyline of this restaurant table consists of just this restaurant. It trumps all others. However, we may have gotten tired of eating there. Or we simply do not like this restaurant ourselves. We would like to search the restaurant table for "best" restaurants, excluding this one. Stratum $s_1(\mathbf{R})$ would do this for us.

We can adapt SFS to compute the top several skyline strata simultaneously. Say we want to compute $s_0(\mathbf{R})$, $s_1(\mathbf{R})$, and $s_2(\mathbf{R})$. SFS can use three windows, one for each stratum. Now if a tuple is found to be dominated by a tuple in window one (a s_0, or skyline, tuple), instead of deleting it, we compare it with tuples in window two. It may belong to s_1. If it is incomparable, we add it to window two as it is a s_1 tuple. Otherwise, we compare it with window three. By the end, we have computed the first three strata. How expensive this is depends on how large $s_0(\mathbf{R})$, $s_1(\mathbf{R})$, and $s_2(\mathbf{R})$ are.

5 Timing Results and Comparisons

In [4, 17], both run experiments over a 100,000 tuple table. Each tuple is 100 bytes, for a 10MB data-set. We run our experiments over a million tuple table. Again, each tuple is 100 bytes, for a 100MB data-set. Each of our tuples consists of ten integer attributes (four bytes each) and a sixty byte string. A page for us is 4096 bytes, so 40 tuples fit per page. We use the integer columns for skyline dimensions. The data was randomly generated, each integer has a value from -MAXINT to MAXINT, the values are uniformally distributed, and the columns are pairwise independent.

Our testing program is written in C++ (Microsoft Visual C++, V6.0). We implement both SFS and BNL in the same code-base for purposes of comparison. As the two algorithms are similar, if-then-else statements switch to specifics for SFS or BNL from the main routine, as needed. We implement the basic BNL algorithm, as discussed in [4]. We do not implement any of the optimizations discussed for BNL, although we note that basic BNL and the optimized versions track quite closely in the experiments in [4]. We ran the experiments on a AMD Athlon 900-MHz PC with 384-MB main memory and a 40-gigabyte disk (7200-rpm, UDMA 100), running Microsoft Windows 2000.

Figure 9 shows timing results versus window size for three versions of SFS: the basic SFS algorithm with the input in nested sorted order as by the query in Figure 6; SFS with entropy sorted input instead (w/E); and SFS with entropy sorted input and applying the projection optimization (w/E,P). For the last, we project out the sixty-byte string of a tuple being added to the window. Thus, for SFS (w/E,P), 100 tuples fit per page in the window.[7] Where the

[7] For the experiments with fewer than the possible ten dimensions, we could have projected out the unused integer

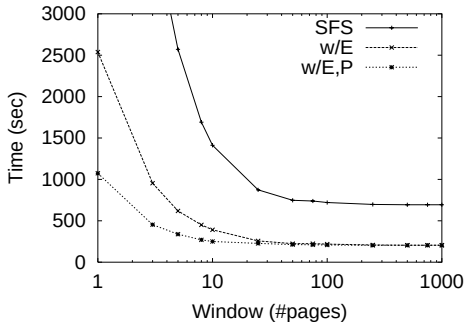

Figure 9: SFS Times (7 dim.).

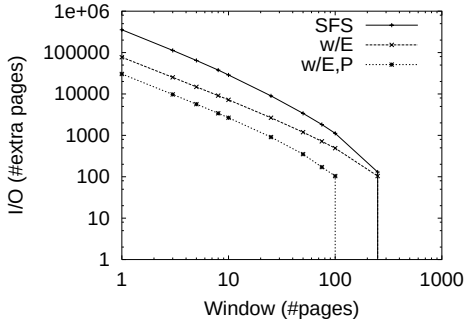

Figure 10: SFS I/O's (7 dim.).

lines level out to horizontal is where the window was large enough the skyline could be found in one pass. The number of skyline tuples with respect to the first seven attributes (dimensions) and our table is 14,081 (or 353 pages at 40 tuples per page, and 141 pages at 100 tuples per page). This happens sooner for SFS (w/E,P) since its window is effectively bigger (in number of tuples). The reason that SFS (w/E) performs better than SFS initially is because it writes fewer pages for subsequent passes because more tuples are eliminated due to the entropy ordering. It is better once the times have leveled because it is less CPU-bound as it eliminates tuples with less CPU overhead (because the window tuples have higher *dn*'s) since a dominating tuple in the window is encountered faster.

The times reported for all SFS experiments *include* the time for pre-sorting. The million tuples were sorted in each case via an external sort routine with the equivalent of a 1,000 page buffer allocation. We treat the sort phase and the filter phase of SFS as two separate operations. This is legitimate, since the two operations would be separately allocated and scheduled by the optimizer in any relational system. It is also reasonable to expect that the

columns too. So for a five dimensional skyline, 200 tuples would have fit per page. However, we unintentionally neglected to do this. If this were corrected, the results for SFS would be that much more improved.

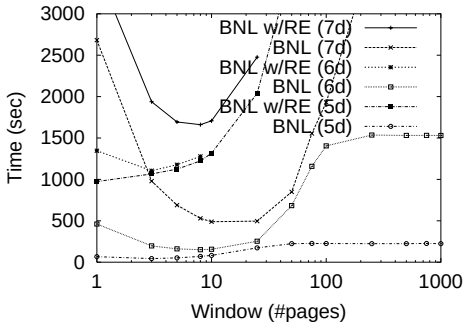

Figure 11: Times for BNL (5, 6, & 7 dim.).

buffer pool allocation for each phase (sort and filter) then might differ. Indeed, the filter phase will not need the size window that would be best for external sorting. Most commercial systems pre-allocate a large portion of buffer space just for sorting operations. Thus the 1,000 page allocation we assumed for sorting is reasonable. With this, nested-sorting the heapfile on the seven attributes for SFS took 57 seconds. Sorting on a single attribute (the tuples' E value, computed on-the-fly) for SFS (w/E) took 37 seconds. These times are reflected in Figure 9. This is another advantage of entropy ordering: Sorting on a single attribute is faster than nested-sorting over a number of attributes.

Figure 10 shows an I/O comparison of the three SFS variants. All the algorithms require 25,000 pages to read the initial heapfile of tuples for the first pass. Thus, we do not include these I/O's in the count. The number of extra pages is the number of pages written cumulatively over all subsequent passes. Each page requires two I/O's: when it is written, and when it is read on the subsequent pass. Where the lines drop down to zero is the point at which the skyline could be computed in one pass. After that point, no additional passes, so no additional pages, were required. SFS and SFS (w/E) obviously reach that point at the same window size. Before that, the fact that SFS (w/E) performs with so fewer I/O's demonstrates the entropy order's effectiveness. SFS (w/E,P) requires fewer I/O's because it is finding more skyline tuples per pass, since more tuples fit into the window. Its drop-off to zero occurs earlier for this reason.

In Figure 11, we plot the performances of BNL over the data-set for skylines of 5, 6, and 7 dimensions, respectively. The five dimensional skyline has 1,651 tuples, the six, 5,357, and the seven, 14,081. We tested two variants of BNL. Each variant is the same algorithm, but different input orders for the tuples. For BNL, the input order is random. (It is the order we generated the data-set in, and the data-set is randomly generated.) For BNL (w/RE), the

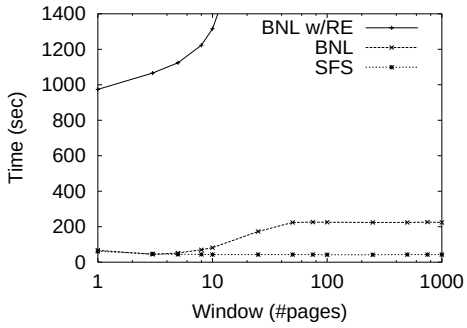

Figure 12: Times for SFS versus BNL (5 dim.).

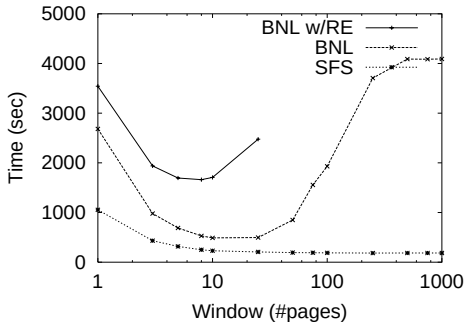

Figure 13: Times for SFS versus BNL (7 dim.).

input order is the tuples sorted by entropy scores, but in *ascending* order (thus, *reverse entropy* ordering). This should be a worse case for BNL since it removes the reduction factor benefit BNL enjoys due to window replacement. BNL's performance in a relational setting for a given data-set and ordering could fall anywhere between these. The lines for BNL (w/RE) stop because we curtailed experiments at larger window allocations for them as they took hours to run.

That BNL is CPU-bound is evident: As the window allocation is increased, at some point BNL's times start rising. This is due to the fact that checking new tuples against the tuples in the window is expensive. When the window is larger, this takes more time.

Note that BNL appears to get substantially worse at higher dimensions. This is not a problem of more dimensions, per se. Number of dimensions is not an operating parameter for either BNL or SFS. Rather, for our experiments, we use the same million-tuple data-set, and add attributes to the the skyline criteria to increase dimensions. This has the effect both that more tuple-pairs become incomparable and the size of the skyline increases. Since there are more skyline tuples to be found, BNL will require more passes. Since more tuples are mutually incomparable, BNL's window and tuple replacement in the window is not as effective. So BNL becomes less effective at eliminating tuples, and the number of pages written for subsequent passes starts to rise.

We compare BNL with SFS. From here on, when we say SFS, we mean SFS (w/E,P). In Figure 12, we compare SFS, BNL, and BNL(w/RE) for computing a five dimensional skyline over the data-set. In Figure 13, we compare them for computing a seven dimensional skyline. In Figure 14 and Figure 15, we compare SFS and BNL on I/O. Again, we do not count the 25,000 pages read for the first pass each makes. The drop-offs to zero represent when the window was sufficient for a single pass, so no additional pages were produced. That SFS drops off

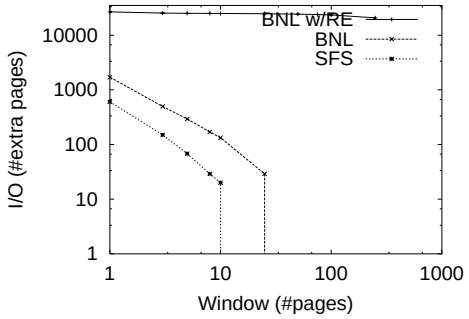

Figure 14: I/O's for SFS versus BNL (5 dim.).

sooner is due to the projection optimization. Note that the slope for SFS is steeper than for BNL. This is significant, as the figures are plotted on logarithmic scale. SFS improves more rapidly in reducing the number of pages read with larger window allocations. SFS makes more efficient use of the window. For BNL (w/RE), I/O performance is horrible. This is because window replacement is greatly diminished for BNL (w/RE), so few tuples are discarded each pass.

We did not test SFS using the last optimization discussed in Section 4, *dimensional reduction*. It is only effective when the ranges of attributes are small. We generated another million tuple database

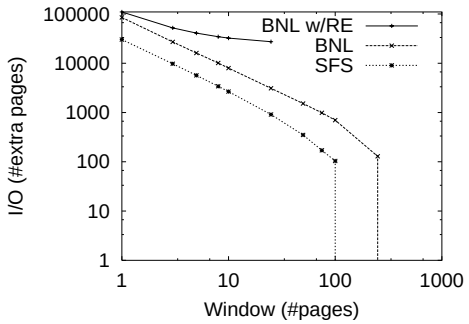

Figure 15: I/O's for SFS versus BNL (7 dim.).

randomly with ranges of 0 to 9 for each integer attribute. For a four-dimensional skyline, and taking the max for the fourth attribute, the sorted table size reduced to 99,826 tuples, so the filter phase of SFS effectively runs on an input 10% of the table's original size.

For SFS adapted for finding skyline strata, it computed s_0, s_1, s_2, and s_3 for four dimensional skyline with respect to the original data-set in 118 seconds, with a window allocation of 500 pages. The s_i's were 460, 1430, 2766, and 4,444 tuples in size, respectively. For a five dimensional skyline, it computed the first four strata in 723 seconds, with a window allocation of 500 pages. They were 1,651, 5,749, 11,879, and 19,020 tuples, respectively.

6 Conclusions and Future Work

There are possible avenues to improve skyline computation performance. There may be ways to speed up checking for dominance in the window to reduce CPU-boundedness. In [4], replacement strategies are discussed for BNL. SFS does not employ replacement, but a certain ordering of tuples in the window, or indexing of window tuples, could increase performance. It might also be possible to incorporate ideas from the divide-and-conquer approach into SFS.[8] In particular, removal of non-skyline tuples could be done during the external sort passes, leaving fewer tuples for the filter passes. Special cases of skyline are known to have good solutions, as for two- and three-dimensional skylines [4]. Perhaps these special cases could be exploited to benefit general skyline computation.

Better provisions to handle anti-correlated data are needed.[9] With anti-correlated attributes as skyline criteria, the size of the skyline can be huge. With 100% anti-correlation, the skyline is the table itself. Currently, both SFS (and BNL) will degenerate into $\lceil |\mathbf{R}|/|\text{Window}| \rceil$ number of passes in this case.

We would like to develop an algorithm to compute efficiently the skyline strata of a table; that is, label each tuple with its stratum number (with respect to the skyline criteria). We want also to extend skyline algorithms to handle more general cases of winnow [6]. We need a better theoretical understanding of skyline, and how it can be used. We want to study which preference queries can be expressed with the skyline operator. In [8], we begin to address the question of how many skyline tuples are likely for a given table **R**. A cardinality estimator for skyline queries is necessary if skyline is to be incorporated into relational engines. The query optimizer's cost model would need to be extended to accommodate skyline queries. A better understanding of the algebra of the skyline operator would be beneficial. For example, a skyline can be computed from subskylines; that is, both "skyline of a_1, a_2" and "skyline of a_3, a_4", can be computed from just "skyline of a_1, a_2, a_3, a_4", but not vice-versa. Finally, it is interesting, and exciting, to note that efficient algorithms for skyline could lead to approaches for faster ways to evaluate queries that use except.

We believe that the skyline operator offers a good start to providing the functionality of preference queries in relational databases, and would be easy for users to employ. We believe that our SFS algorithm for skyline offers a good start to incorporating the skyline operator into relational engines, and hence, into the relational repertoire, effectively and efficiently.

[8] The skyline of the union of two relations can be computed as the skyline of the union of the skylines of both relations.

[9] Also, while we expect that SFS with entropy scoring will work well in these cases too, we must further test SFS to verify this.

References

[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *Proceedings of SIGMOD*, pages 297–306, 2000.

[2] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *JACM*, 25(4):536–543, 1978.

[3] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of PODS*, pages 78–86, 1997.

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

[5] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proceedings of VLDB*, pages 158–169. Morgan Kaufmann, 1998.

[6] J. Chomicki. Querying with intrinsic preferences. In *Proceedings of EDBT*, 2002.

[7] P. Godfrey. Cardinality estimation of skyline queries. Technical Report CS-2002-03, Computer Science, York University, Toronto, Ontario, Canada, Oct. 2002.

[8] P. Godfrey. Cardinality estimation of skyline queries: Harmonics in data. Submitted, 2002.

[9] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of SIGMOD*, pages 259–270, 2001.

[10] N. Katayama and S. Satoh. Nearest neighbor queries. In *Proceedings of SIGMOD*, pages 71–79, 1995.

[11] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of SIGMOD*, pages 369–380, 1997.

[12] W. Kießling. Foundations of preferences in database systems. In *Proceedings of the 28*th *Conference on Very Large Databases (VLDB)*, Aug. 2002.

[13] W. Kießling and G. Köstler. Preference SQL: Design, implementation, experiences. In *Proceedings of the 28*th *Conference on Very Large Databases (VLDB)*, Aug. 2002.

[14] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28*th *Conference on Very Large Databases (VLDB)*, Aug. 2002.

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

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

[17] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proceedings of VLDB*, pages 301–310, 2001.

[18] *Zagat Toronto Restaurants.* Zagat Survey, 2002.