The Skyline Operator[*]

Stephan Börzsönyi[1] Donald Kossmann[2] Konrad Stocker[1]

[1]Universität Passau
D-94030 Passau, Germany
⟨*lastname*⟩@db.fmi.uni-passau.de

[2]Technische Universität München
D-81667 München, Germany
kossmann@in.tum.de

Abstract

We propose to extend database systems by a *Skyline* operation. This operation filters out a set of *interesting* points from a potentially large set of data points. A point is interesting if it is not dominated by any other point. For example, a hotel might be interesting for somebody traveling to Nassau if no other hotel is both cheaper and closer to the beach. We show how SQL can be extended to pose Skyline queries, present and evaluate alternative algorithms to implement the *Skyline* operation, and show how this operation can be combined with other database operations (e.g., *join* and *Top N*).

1 Introduction

Suppose you are going on holiday to Nassau (Bahamas) and you are looking for a hotel that is cheap and close to the beach. Unfortunately, these two goals are complementary as the hotels near the beach tend to be more expensive. The database system at your travel agents' is unable to decide which hotel is best for you, but it can at least present you all *interesting* hotels. Interesting are all hotels that are not worse than any other hotel in both dimensions. We call this set of interesting hotels the *Skyline*. From the Skyline, you can now make your final decision, thereby weighing your personal preferences for price and distance to the beach.

Computing the Skyline is known as the maximum vector problem [KLP75, PS85]. We use the term Skyline because of its graphical representation (see below). More formally, the Skyline is defined as those points which are not dominated by any other point. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension. For example, a hotel with *price* $=$ $50 and *distance* $=$ 0.8 miles dominates a hotel with *price* $=$ $100 and *distance* $=$ 1.0 miles.

Figure 1 shows the Skyline of cheap hotels near the beach for a sample set of hotels. A travel agency is one application for which a Skyline operation would be useful. Clearly, many other applications in the area of decision support can be found; e.g., finding good salespersons which have low salary. A Skyline operation can also be very useful for database visualization. The Skyline of Manhattan, for instance, can be computed as the set of buildings which are high and close to the Hudson river. In other words, a building dominates another building if it is higher, closer to the river, and has the same x position (buildings next to each other can both be seen). This is shown in Figure 2.

One of the nice properties of the Skyline of a set $\mathcal{M}$ is that for any monotone scoring function $\mathcal{M} \rightarrow \mathbf{R}$, if $p \in \mathcal{M}$ maximizes that scoring function, then p is in the Skyline. In other words, no matter how you weigh your personal preferences towards price and distance of hotels, you will find your favorite hotel in the Skyline. In addition, for every point p in the Skyline, there exists a monotone scoring function such that p maximizes that scoring function. In other words, the Skyline does not contain any hotels which are nobody's favorite.

As mentioned earlier, the Skyline operation has been studied in previous work [KLP75, PS85]. That work, however, assumes that the whole set of points fits into memory and that this set is not the result of a query (e.g., a join). In this work, we show how the Skyline operation can be integrated into a database

[*]This research is supported by the German National Research Foundation under contract DFG Ke 401/7-1

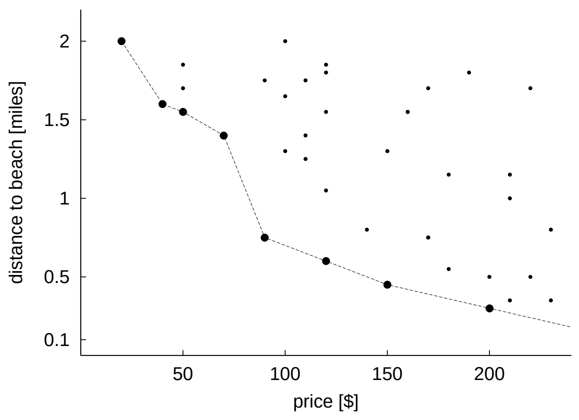


Figure 1: Skyline of Hotels

Figure 2: Skyline of Manhattan

system. We will first describe possible SQL extensions in order to specify Skyline queries. Then, we will present and evaluate alternative algorithms to compute the Skyline; it will become clear that the original algorithm of [KLP75, PS85] has terrible performance in the database context. We will also briefly discuss how standard index structures such as B-trees and R-trees can be exploited to evaluate Skyline queries. In addition, we will show how the Skyline operation can interact with other query operations; e.g., joins, group-by, and Top N. At the end, we will discuss related work, give conclusions, and make suggestions for future work.

2 SQL Extensions

In order to specify Skyline queries, we propose to extend SQL's `SELECT` statement by an optional `SKYLINE OF` clause as follows:

```
SELECT ...  FROM ...  WHERE ...
GROUP BY ...  HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX | DIFF], ..., dm [MIN | MAX | DIFF]
ORDER BY ...
```

$d_1, ..., d_m$ denote the dimensions of the Skyline; e.g., *price*, *distance to the beach*, or *rating*. `MIN`, `MAX`, and `DIFF` specify whether the value in that dimension should be minimized, maximized, or simply be different. For example, the *price* of a hotel should be minimized (`MIN` annotation) whereas the rating should be maximized (`MAX` annotation). In our Skyline of Manhattan example, two buildings that have different x coordinates can both be seen and therefore both may be part of the skyline; as a result, the x dimension is listed in the `SKYLINE OF` clause of that query with a `DIFF` annotation. The optional `DISTINCT` specifies how to deal with duplicates (described below).

The semantics of the `SKYLINE OF` clause are straightforward. The `SKYLINE OF` clause is executed after the `SELECT ... FROM ... WHERE ... GROUP BY ... HAVING...` part of the query, but before the `ORDER BY` clause and possibly other clauses that follow (e.g., `STOP AFTER` for Top N [CK97]). The `SKYLINE OF` clause selects all *interesting* tuples; i.e., tuples which are not dominated by any other tuple. Extending our definition from the introduction tuple $p = (p_1, \dots, p_k, p_{k+1}, \dots, p_l, p_{l+1}, \dots, p_m, p_{m+1}, \dots, p_n)$ dominates tuple $q = (q_1, \dots, q_k, q_{k+1}, \dots, q_l, q_{l+1}, \dots, q_m, q_{m+1}, \dots, q_n)$ for a Skyline query

`SKYLINE OF` d_1 `MIN`, $\dots$, d_k `MIN`, d_{k+1} `MAX`, $\dots$, d_l `MAX`, d_{l+1} `DIFF`, $\dots$, d_m `DIFF`

if the following three conditions hold:

$$\begin{array}{lll} p_i \le q_i & \text{for all} & i = 1, \dots, k \\ p_i \ge q_i & \text{for all} & i = (k+1), \dots, l \\ p_i = q_i & \text{for all} & i = (l+1), \dots, m \end{array}$$

```
SELECT *
FROM Hotels
WHERE city = 'Nassau'
SKYLINE OF price MIN, distance MIN;
```

Query 1

```
SELECT *
FROM Buildings
WHERE city = 'New York'
SKYLINE OF distance MIN, height MAX,
           x DIFF;
```

Query 2

```
SELECT e.name, e.salary, sum(s.volume) as volume
FROM Emp e, Sales s
WHERE e.id = s.repr AND s.year = 1999
GROUP BY e.name, e.salary
SKYLINE OF e.salary MIN, volume MAX;
```

Query 3

```
SELECT name, distance,
       (CASE WHEN price ≤ 50 THEN 'cheap'
             WHEN price > 50 THEN 'exp') AS pcat
FROM Hotels
WHERE city = 'Nassau'
SKYLINE OF pcat MIN, distance MIN;
```

Query 4

Figure 3: Example Skyline Queries

If $p_i = q_i$ for all $i = 1, \ldots, m$, then p and q are incomparable and may both be part of the Skyline if no `DISTINCT` is specified. With `DISTINCT`, either p or q are retained (the choice of which is unspecified). The values of the attributes $d_{m+1}, \ldots, d_n$ are irrelevant for the Skyline computation, but these attributes are of course part of the tuples of the Skyline (i.e., there is no implicit projection). Note that it does not matter in which order the dimensions are specified in the `SKYLINE OF` clause; for ease of presentation, we put the `MIN` dimensions first and the `DIFF` dimensions last. In addition, a one-dimensional Skyline is equivalent to a *min, max,* or *distinct* SQL query without a `SKYLINE OF` clause. Furthermore, note that *dominance* is a transitive relation; if p dominates q and q dominates r, then p also dominates r. Transitivity is an important property to implement the Skyline operation (Section 3).

Figure 3 contains four example Skyline queries. The first two queries are our example queries from the introduction; i.e., cheap hotels near the beach in Nassau and the Skyline of Manhattan. The third query asks for salespersons who were very successful in 1999 and have low salary; these people might be eligible for a raise. Like the first query, the fourth query asks for cheap hotels near the beach in Nassau – this time, however, at most two hotels are returned because the query specifies *price categories*; within each *price category* the user is only interested for the hotel with the smallest distance to the beach. Naturally, attributes which are specified in the `SKYLINE OF` clause may also be used in any other clause. To eliminate outrageously expensive hotels, for example, the `WHERE` clause of the first query of Figure 3 could be extended by a *price* < 250 predicate.

3 Implementation of the Skyline Operator

Our approach to implement Skyline queries is to extend an existing (relational, object-oriented or object-relational) database system with a new logical operator that we refer to as the Skyline operator. The Skyline operator encapsulates the implementation of the `SKYLINE OF` clause. The implementation of other operators (e.g., *join*) need not be changed. According to the semantics of Skyline queries, the Skyline operator is typically executed after *scan, join,* and *group-by* operators and before a final *sort* operator, if the query has an `ORDER BY` clause. (In Section 4, we will discuss exceptions to this rule.) As a result, only simple modifications to the parser and query optimizer are required and integrating the Skyline operator into a traditional SQL query processor is extremely simple.

Just like *join* and most other logical operators, there are several different (physical) ways to implement the Skyline operator. In this section, we will describe seven variants: three variants based on a block-nested-loops algorithm; three variants based on divide-and-conquer; and one special variant for two-dimensional Skylines. Furthermore, we will show how Skyline queries can be implemented *on top of* a relational database system, without changing the database system at all; it will become clear, however, that this approach performs very poorly.

3.1 Translating a Skyline Query into a Nested SQL Query

We will begin and show how Skyline queries can be implemented on top of a relational database system by translating the Skyline query into a nested SQL query. We will demonstrate this using our hotel Skyline query from Figure 3. This query is equivalent to the following standard SQL query:

$\langle h_1,\ \$50,\ 3.0\,\text{miles}\rangle$
$\langle h_2,\ \$51,\ 5.0\,\text{miles}\rangle$
$\langle h_3,\ \$52,\ 4.0\,\text{miles}\rangle$
$\langle h_4,\ \$53,\ 2.0\,\text{miles}\rangle$

Figure 4: 2-d Skyline: Sorting Works

$\langle h_1,\ \$50,\ 3.0\,\text{miles},\ ***\rangle$
$\langle h_2,\ \$51,\ 5.0\,\text{miles},\ ****\rangle$
$\langle h_3,\ \$52,\ 4.0\,\text{miles},\ ***\rangle$
$\langle h_4,\ \$53,\ 2.0\,\text{miles},\ ***\rangle$

Figure 5: 3-d Skyline: Sorting Does Not Work

```
SELECT *
FROM   Hotels h
WHERE  h.city = 'Nassau' AND NOT EXISTS(
       SELECT *
       FROM   Hotels h1
       WHERE  h1.city = 'Nassau' AND h1.distance <= h.distance AND
              h1.price <= h.price AND
              (h1.distance < h.distance OR h1.price < h.price));
```

Thus, we can very well express in SQL that we are interested in hotels that are not dominated by other hotels. However, this approach shows very poor performance for a number of reasons:

- Essentially, this approach corresponds to the naive "nested-loops" way to compute the Skyline because this query cannot be unnested [GKG$^+$97, BCK98]; as we will see in the following subsections, we can do much better.
- If the Skyline query involves a join or group-by (e.g., the third query of Figure 3), this join or group-by would have to be executed as part of the outer query and as part of the subquery.
- As we will see in Section 4 the Skyline operation can be combined with other operations (e.g., *join* or *Top N*) in certain cases, resulting in little additional cost to compute the Skyline.

As a result, we propose to extend the database system and integrate a Skyline operator into the system. We focus on such an approach in the remainder of this paper.

3.2 Two-dimensional Skyline Operator

As mentioned in Section 2, a one-dimensional Skyline is trivial because it is equivalent to computing the *min, max,* or *distinct*. Computing the Skyline is also very easy if the `SKYLINE OF` clause involves only two dimensions. A two-dimensional Skyline can be computed by sorting the data. If the data is topologically sorted according to the two attributes of the `SKYLINE OF` clause, the test of whether a tuple is part of the Skyline is very cheap: you simply need to compare a tuple with its predecessor. More precisely, you need to compare a tuple with the last previous tuple which is part of the Skyline. Figure 4 illustrates this approach. h_2 can be eliminated because it is dominated by h_1, its predecessor. Likewise, h_3 can be eliminated because it is dominated by h_1, its predecessor after h_2 has been eliminated.

If sorting needs to be carried out in two (or more passes) because the data does not fit into main memory, then tuples can be eliminated while generating each run and in the *merge* phase of the sort. Eliminating tuples during the run generation (i.e., as part of replacement selection or quicksort) makes the runs smaller and therefore possibly saves a great deal of disk I/O. Doing such an early elimination of tuples is analogous to "early aggregation" which is used to improve the performance of group-by operations [BD83, Lar97].

Figure 5 shows why sorting does not work if the Skyline involves more than two dimensions. In this example, we are interested in hotels with a low price, a short distance to the beach, and a high rating (many stars). The only hotel which can be eliminated is h_3: h_3 is dominated by h_1, but h_1 is not h_3's direct predecessor. In this example, there is just one hotel between h_1 and h_3; in general, however, there might be many hotels so that sorting does not help. There are special algorithms to deal with three-dimensional Skylines [KLP75], but for brevity we will not discuss such algorithms in this work.

3.3 Block-nested-loops Algorithm

The naive way to compute the Skyline is to apply a nested-loops algorithm and compare every tuple with every other tuple. This is essentially what happens if a Skyline query is implemented on top of a database system as described in Section 3.1.[1] The naive nested-loops algorithm can be applied to any Skyline query (not just two dimensional), but it obviously is very inefficient. In this section, we present an algorithm that is significantly faster; rather than considering a tuple at a time, this algorithm produces a block of Skyline tuples in every iteration.

3.3.1 Basic Block-nested-loops Algorithm

Like the naive nested-loops algorithm, the block-nested-loops algorithm repeatedly reads the set of tuples. The idea of this algorithm is to keep a *window* of incomparable tuples in main memory. When a tuple p is read from the input, p is compared to all tuples of the window and, based on this comparison, p is either eliminated, placed into the window or into a temporary file which will be considered in the next iteration of the algorithm. Three cases can occur:

1. p is dominated by a tuple within the window. In this case, p is eliminated and will not be considered in future iterations. Of course, p need not be compared to all tuples of the window in this case.

2. p dominates one or more tuples in the window. In this case, these tuples are eliminated; that is, these tuples are removed from the window and will not be considered in future iterations. p is inserted into the window.

3. p is incomparable with all tuples in the window. If there is enough room in the window, p is inserted into the window. Otherwise, p is written to a temporary file on disk. The tuples of the temporary file will be further processed in the next iteration of the algorithm. When the algorithm starts, the first tuple will naturally be put into the window because the window is empty.

At the end of each iteration, we can output tuples of the window which have been compared to all tuples that have been written to the temporary file; these tuples are not dominated by other tuples (i.e., they are part of the Skyline) and they do not dominate any tuples which are still under consideration. Specifically, we can output and ignore for further processing those tuples which were inserted into the window when the temporary file was empty; i.e., at the beginning of the iteration. The other tuples of the window can be output (if they are not eliminated) *during* the next iteration. The earlier a tuple has been inserted into the window, the earlier the tuple can be output during the next iteration. In order to keep track when a tuple of the window may be output, we assign a *timestamp* to each tuple in the window and to each tuple in the temporary file. This timestamp is a simple counter that records in which order tuples were inserted into the window and temporary file. If we read a tuple from the temporary file with timestamp t, we can output all tuples from the window with timestamp smaller than t. (This timestamping also guarantees that the algorithm terminates and that two tuples are never compared twice.) The full algorithm is given in Appendix A.

Obviously this algorithm works particularly well if the Skyline is small. In the best case, the Skyline (i.e., the result) fits into the window and the algorithm terminates in one or two iterations; thus, the best case complexity is of the order of $\mathcal{O}(n)$; n being the number of tuples in the input. In the worst case, the complexity is of the order of $\mathcal{O}(n^2)$. The asymptotic complexity is the same as for the naive nested-loops algorithm, but the block-nested-loops algorithm shows much better I/O behavior than the naive nested-loops algorithm in a similar way as the block-nested-loops join algorithm is better than a naive nested-loops join [HCLS97].

3.3.2 Variants of the Basic Algorithm

Maintaining the Window as a Self-organizing List A great deal of time in the basic algorithm is spent for comparing a tuple with the tuples in the window. To speed up these comparisons, we propose to organize the window as a *self-organizing list*. When tuple w of the window is found to dominate (i.e., eliminate) another tuple, then w is moved to the beginning of the window. As a result, the next tuple from the input will be compared to w first. This variant is particularly attractive if the data is skewed; i.e., if there are a couple of *killer* tuples which dominate many other tuples and a significant number of *neutral* tuples which are part of the Skyline, but do not dominate other tuples.

[1] In fact, the nested SQL query approach is even worse than the naive nested-loops algorithm because a tuple cannot be eliminated in the “inner loop.”

Replacing Tuples in the Window As another variant, we propose to try to keep the *most dominant* set of tuples in the window. As an example, assume that the window contains the following two hotels:

$\langle h_1, \$50, 1.0 \text{ mile}\rangle$ $\langle h_2, \$59, 0.9 \text{ mile}\rangle$

Also assume that the next hotel to be considered is

$\langle h_3, \$60, 0.1 \text{ mile}\rangle$

h_3 is incomparable to both h_1 and h_2. If the window has a capacity of two hotels, the basic algorithm would write h_3 into the temporary file, leaving the window unchanged. In this example, however, it is easy to see that a window containing h_1 and h_3 is likely to eliminate more hotels than the old window of h_1 and h_2 because h_3 is only slightly more expensive but significantly closer to the beach than h_2. In other words, h_3 should *replace* h_2 in the window and h_2 should be written to the temporary file.

There are many conceivable replacement policies. One option is to extend our *self-organizing list* variant and implement an LRU replacement policy. In this work, we use a replacement policy that is based on the *area* covered by a tuple in order to decide which tuples should be kept in the window. For instance, h_3 is much better in terms of *price* $*$ *distance* than h_2.

Implementing such a replacement policy does not come for free. First, there are additional CPU costs to determine the replacement victim. Second, a tuple which is replaced from the window is written to the end of the temporary file and needs to be compared to other tuples placed before it in the temporary file again; as a result, with tuple replacement is it possible that two tuples are compared twice. We will study the tradeoffs of the replacement variant in more detail in Section 5.

3.4 Divide and Conquer Algorithm

We will now turn to the divide-and-conquer algorithm of [KLP75, PS85] and our proposed extensions to make that algorithm more efficient in the database context (i.e., limited main memory). This algorithm is theoretically the best known algorithm for the worst case. In the worst case, its asymptotic complexity is of the order of $\mathcal{O}(n * (\log n)^{d-2}) + \mathcal{O}(n * \log n)$; n is the number of input tuples and d is the number of dimensions in the Skyline. Unfortunately, this is also the complexity of the algorithm in the best case; so, we expect this algorithm to outperform our block-nested-loops algorithm in bad cases and to be worse in good cases.

3.4.1 Basic Divide and Conquer Algorithm

The basic divide-and-conquer algorithm of [KLP75, PS85] works as follows:

1. Compute the median m_p (or some approximate median) of the input for some dimension d_p (e.g., *price*). Divide the input into two partitions. P_1 contains all tuples whose value of attribute d_p is better (e.g., less) than m_p. P_2 contains all other tuples.

2. Compute the Skylines S_1 of P_1 and S_2 of P_2. This is done by recursively applying the whole algorithm to P_1 and P_2; i.e., P_1 and P_2 are again partitioned. The recursive partitioning stops if a partition contains only one (or very few) tuples. In this case, computing the Skyline is trivial.

3. Compute the overall Skyline as the result of *merging* S_1 and S_2. That is, eliminate those tuples of S_2 which are dominated by a tuple in S_1. (None of the tuples in S_1 can be dominated by a tuple in S_2 because a tuple in S_1 is better in dimension d_p than every tuple of S_2.)

Most challenging is Step 3. The main trick of this step is shown in Figure 6. The idea is to partition both S_1 and S_2 using an (approximate) median m_g for some other dimension d_g, with $d_g \neq d_p$. As a result, we obtain four partitions: $S_{1,1}, S_{1,2}, S_{2,1}, S_{2,2}$. $S_{1,i}$ is better than $S_{2,i}$ in dimension d_p and $S_{i,1}$ is better than $S_{i,2}$ in dimension d_g ($i = 1, 2$). Now, we need to merge $S_{1,1}$ and $S_{2,1}$, $S_{1,1}$ and $S_{2,2}$, and $S_{1,2}$ and $S_{2,2}$. The beauty is that we need not merge $S_{1,2}$ and $S_{2,1}$ because the tuples of these two sets are guaranteed to be incomparable. Merging $S_{1,1}$ and $S_{2,1}$ (and the other pairs) is done by recursively applying the *merge* function. That is, $S_{1,1}$ and $S_{2,1}$ are again partitioned. The recursion of the *merge* function terminates if all

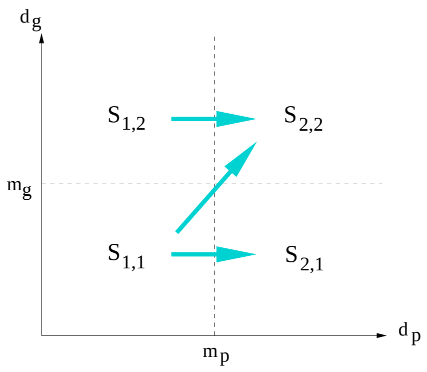


Figure 6: Basic Merge

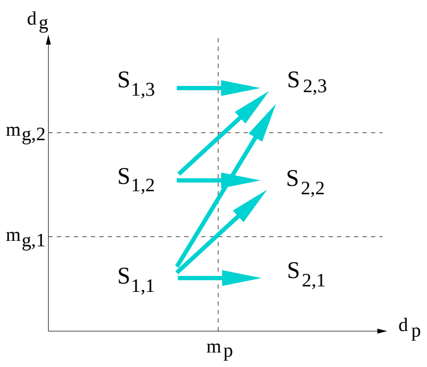


Figure 7: 3-way Merge

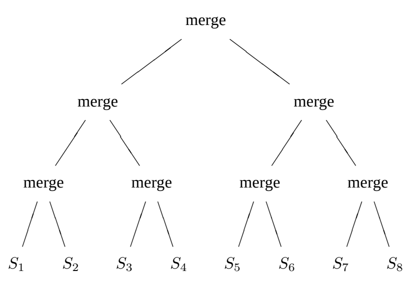


Figure 8: Bushy Merge Tree

dimensions have been considered or if one of the partitions is empty or contains only one tuple; in all these cases the *merge* function is trivial. A full definition of this algorithm can be found in Appendix B. The algorithm has also been described in great detail in [PS85].

3.4.2 Extensions to the Basic Algorithm

M-way Partitioning If the input does not fit into main memory, the basic algorithm shows terrible performance. The reason is that the input is read, partitioned, written to disk, reread to be partitioned again, and so on several times until a partition fits into main memory. One may argue that main memories are becoming larger and larger; at the same time, however, databases are becoming larger and larger and more and more concurrent queries must be executed by a database server so that the available main memory per query is limited. As a result, no database vendor today will be willing to implement an algorithm that relies on the fact that all the data fit into memory.

Fortunately, the I/O behavior of the basic algorithm can be improved quite easily. The idea is to divide into m partitions in such a way that every partition is expected to fit into memory. Instead of the median, α-quantiles are computed in order to determine the partition boundaries. If a partition does not fit into memory, it needs to be partitioned again; this is analogous to recursive partitioning as needed, e.g., for hash join processing [Gra93].

m-way partitioning can be used in the first step of the basic algorithm as well as in the third step. In the first step, m-way partitioning is used to produce m partitions $P_1, \ldots, P_m$ so that each P_i fits into memory and S_i, the Skyline of P_i, can be computed in memory using the basic algorithm. The final answer is produced in the third step by merging the S_i pairwise. Within the *merge* function, m-way partitioning is applied so that all sub-partitions can be merged in main memory; i.e., all sub-partitions should occupy at most half the available main memory. Figure 7 shows which sub-partitions need to be merged if a three-way sub-partitioning has been applied in the *merge* function.

As shown in Figure 8, we propose to apply the *merge* function to the initial partitions in a *bushy* way. As a result, the volume of data that must be read and rewritten to disk between different merge steps is minimized. For instance, in Figure 8 the tuples of S_1 are only involved in $\log m$ merge steps; these tuples would be involved in $m - 1$ merge steps if all the merging were carried out in a *left-deep* way. Like all other algorithms, we define the full divide-and-conquer algorithm with m-way partitioning in Appendix C.

Early Skyline We propose another very simple extension to the divide-and-conquer algorithm in situations in which the available main memory is limited. This extension concerns the first step in which the data is partitioned into m partitions. We propose to carry out this step as follows:

1. Load a large block of tuples from the input; more precisely, load as many tuples as fit into the available main-memory buffers.

2. Apply the basic divide-and-conquer algorithm to this block of tuples in order to immediately eliminate tuples which are dominated by others. We refer to this as an "Early Skyline;" it is essentially the same kind of prefiltering that is applied in the sorting-based algorithm of Section 3.2.

hotel	*price*
h_1	\$25
h_3	\$27
h_{25}	\$30
h_2	\$35
h_{35}	\$40
h_{17}	\$70
...	

hotel	*distance*
h_{17}	0.1 miles
h_2	0.2 miles
h_{35}	0.3 miles
h_{25}	0.3 miles
h_1	0.7 miles
h_3	1.0 miles
...	

Figure 9: Ordered Indices (B-trees) for the Hotel Query

3. Partition the remaining tuples into m partitions.

The tradeoffs of this approach are quite simple. Clearly, applying an Early Skyline incurs additional CPU cost, but it also saves I/O because less tuples need to be written and reread in the partitioning steps. In general an Early Skyline is attractive if the Skyline is *selective*; i.e., if the result of the whole Skyline operation is small.

4 Other Skyline Algorithms

4.1 Using B-trees

To compute the Skyline, it is also possible to use an ordered index; e.g., a B-tree. One way to use an ordered index for a two-dimensional Skyline is to scan through the whole index, get the tuples in sorted order and filter out the tuples of the Skyline as described in Section 3.2. In the following, we will describe how ordered indices can be used without scanning through the whole index.

Let us go back to our favorite query that asks for cheap hotels that are close to the beach. Let us also assume that we have two ordered indices: one on *hotel.price* and one on *hotel.distance*, as shown in Figure 9. We can use these indices in order to find a superset of the Skyline. We do so by adapting the first step of Fagin's A_0 algorithm used for merging scores from multi-media databases [Fag96]. In other words, we scan simultaneously through both indices and stop as soon as we have found the first match. In the example of Figure 9, h_2 is the first match.

Now, we can draw the following conclusions:

- h_2 is definitely part of the Skyline.
- Any hotel which has not yet been inspected (i.e., below h_2 in both indices) is definitely not part of the Skyline because it is dominated by h_2. One example is h_{35}.
- All other hotels may or may not be part of the Skyline. To determine which of these hotels are part of the Skyline, we need to execute one of the algorithms of Section 3. In Figure 9, for instance, h_1, h_{17}, and h_{25} are part of the Skyline whereas h_3 is not because it is dominated by h_1.

This algorithm can also be applied if the query has predicates in its `WHERE` clause. In this case, we need to search through the indices until we find the first match that qualifies these predicates. In the presence of predicates, the algorithm becomes less attractive because it takes longer to find a match. In general, this algorithm is attractive if the result of the Skyline is very small and a first match (like h_2) can be found very quickly. This algorithm can also be generalized to Skylines with more than two dimensions. In this case, a match for *all* indices is needed; again, the algorithm becomes less effective because typically the match is found later. Furthermore, the applicability of this algorithm is limited if the Skyline of the result of a join or group-by must be computed.

4.2 Using an R-tree

To use an R-tree to compute the Skyline of cheap hotels near the beach, we exploit the following fact:

Given a hotel h, we need not search in any branches of the R-tree which are guaranteed to contain only hotels that are dominated by h.

For example, if we know that there is a hotel that costs \$30 and is located 1.0 miles from the beach, then we need not consider any branches in the R-tree which include, for instance, hotels in the price range of (\$40, \$60) and distance range of (2.0 miles, 3.5 miles). As a consequence, the idea is to traverse the R-tree in a depth first way and *prune* branches of the R-tree with every new hotel found. A related branch-and-bound approach to traverse an R-tree has also been proposed for nearest-neighbor search (e.g., [RKV95]). We plan to investigate different heuristics to select branches of the R-tree for the depth-first search in future work.

Obviously, an R-tree can only be used if the R-tree involves all dimensions specified in the `SKYLINE OF` clause of the query; in addition, the R-tree may involve other attributes which are not part of the `SKYLINE OF` clause. Like B-trees, the use of R-trees is particularly attractive if the Skyline is small and becomes less attractive in the presence of predicates and with a growing number of dimensions in the Skyline. Also, the applicability of R-trees is limited for join and group-by queries. (Essentially, these limitations hold for any application of indices.)

4.3 Skyline and Joins

4.3.1 Pushing the Skyline Operator *Through* a Join

As mentioned in Section 2, the Skyline operator is logically applied after joins and group-bys. However, if a join (or group-by) is *non-reductive* (as defined in [CK97]), then the Skyline operator may also be applied *before* the join. Applying a Skyline operator before a join is attractive because the Skyline operator reduces the size of the result and, therefore, a Skyline operator before a join makes the join cheaper. Also, non-reductive joins tend to increase the size of the result (the tuples get wider) so that the Skyline operator itself becomes cheaper if it is pushed through the join. Furthermore, pushing the Skyline operator through a join might make it possible to use an index to compute the Skyline, as described in Sections 4.1 and 4.2.

As an example, consider the following query which asks for young employees with high salary and their department information:

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno = d.dno
SKYLINE OF e.age MIN, e.salary MAX;
```

Assuming that every employee works in a deptartment, every employee qualifies for the join. (This is what non-reductiveness means.) In this example, it is easy to see why the Skyline of employees can be computed before the join and why doing so is beneficial. Extending the query optimizer to consider such a Skyline push-down is also fairly easy; essentially, the same measures as described in [CK97] for Top N queries can be applied.

4.3.2 Pushing the Skyline Operator *Into* a Join

Let us now consider the following query which asks for Emps with high salary that work in a Dept with low budget:

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno = d.dno
SKYLINE OF d.budget MIN, e.salary MAX;
```

Now assume that we have three Emps that work in Dept 23: Roger with 200K salary, Mary with 400K salary, and Phil with 100K salary. Without knowing the budget of Dept 23, we can immediately eliminate Roger and Phil. That is we can compute the

```
salary MAX, dno DIFF
```

Skyline directly on the Emps before the join, then compute the join and the overall Skyline. This observation leads to the following approach:

1. Sort the `Emp` table by `dno, salary DESC`, thereby eliminating `Emps` that have a lower salary than another `Emp` with the same `dno`.
2. Sort the `Dept` table by `dno`.
3. Apply a merge join.
4. Compute the Skyline using any approach of Section 3.

In effect, we apply an "Early Skyline" with almost no additional overhead as part of a sort-merge join. Therefore, carrying out this approach is a no-loss situation if a sort-merge join is attractive.

4.4 Skyline and Top N

As mentioned in Section 2, `ORDER BY` and *Top N* are applied logically after the Skyline operator in a query. However, if the Top N operation is based on the same or a subset of the columns used in the Skyline operation, then the computation of the Skyline and Top N operations can be combined. For instance, to compute the N cheapest hotels which are interesting in terms of *price*, *distance*, and maybe some other criteria, we could proceed as follows:

1. Sort all hotels by price (i.e., by the criterion of the Top N operation).
2. Scan through the sorted list of hotels. For each hotel, test whether it is interesting by comparing that hotel to all hotels that have the same or lower price.
3. Stop as soon as N hotels have been produced.

This algorithm can be applied independently of the number of dimensions used in the Top N and Skyline operations. The only restriction is that the Top N dimensions must be a subset of the dimensions of the Skyline operation. The reason why this approach is attractive is that in Step 2, a hotel need only be compared to hotels which have a lower price and are part of the Skyline or to hotels that have the same price; in other words, $\mathcal{O}(N)$ comparisons are sufficient for each hotel in Step 2. If N is small and N hotels fit into memory (this is the typical situation for Top N queries), Step 2 can be carried out with a single scan through the sorted list of hotels and the cost of the combined Skyline and Top N operation is dominated by Step 1. If there is an ordered index on *hotels.price*, Step 1 might become very cheap, too, so that the whole combined Skyline and Top N operation can be executed much faster than any of the Skyline algorithms presented in Section 3. If no such ordered index exists or using such an index is expensive because the index is not clustered, then range partitioning instead of sorting can be applied, as proposed in [CK98]. In other words, the combined Skyline and Top N algorithm is first applied to, say, all hotels which cost less than \$100. If N or more hotels have been produced, we are done; otherwise, we also need to apply the combined Skyline and Top N algorithm to hotels in the price range of (\$100, \$120), and so on. As described in [CK98], such an approach helps significantly to avoid wasted work for sorting.

5 Performance Experiments and Results

In this section, we study the performance of the alternative Skyline algorithms presented in Section 3. For our experiments, we use different kinds of synthetic databases (correlated and uncorrelated). Furthermore, we vary the number of dimensions in a Skyline query and the available main memory buffers.

5.1 Experimental Environment

All our experiments are carried out on a Sun Ultra Workstation with a 333 MHz processor and 128 MB of main memory. The operating system is Solaris 7. The benchmark databases and intermediate query results are stored on a 9GB Seagate disk drive with 7200 rpm and 512K disk cache. For our experiments, we implemented all algorithms of Section 3 in C++. Specifically, we implemented the following variants:

Sort: two-phase sorting with an "Early Skyline," as described in Section 3.2. Runs are generated using quicksort. As mentioned in Section 3.2, this approach is only applicable for two-dimensional Skylines;

BNL-basic: the basic block-nested-loops algorithm (Section 3.3.1);

BNL-sol: block-nested-loops; the window is organized as a self-organizing list (Section 3.3.2);

BNL-solrep: block-nested-loops; the windows is organized as a self-organizing list and tuples in the window are replaced (Section 3.3.2);

D&C-basic: Kung et al.'s basic divide-and-conquer algorithm (Section 3.4.1);

D&C-mpt: divide-and-conquer with m-way partitioning (Section 3.4.2);

D&C-mptesk: divide-and-conquer with m-way partitioning and "Early Skyline" (Section 3.4.2).

Furthermore, we run Skyline queries using a commercial relational database systems using the approach described in Section 3.1. The RDBMS is installed on the same machine. The results of the RDBMS are not directly comparable to the other results because the RDBMS is a significantly more complex system which, among others, involves transaction processing. Nevertheless, we present these results because they demonstrate the need to integrate Skyline processing into a database system, rather than implementing it on top of it.

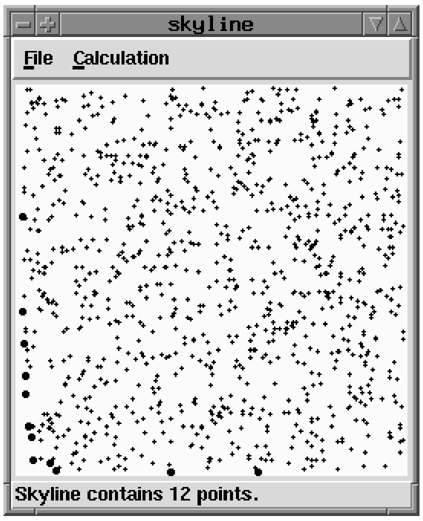
In none of our experiments reported here we make use of indices. We did experiment with different kinds of indices for the RDBMS, but in most cases the presence of indices increased the running time of the RDBMS for our Skyline benchmark queries. Only in very simple cases did indices help the RDBMS, but in such cases our approaches to use indices (Sections 4.1 and 4.2) would have done much better than the RDBMS in exploiting these indices.

If not stated otherwise, each benchmark database contains 100,000 tuples (10MB). A tuple has d attributes of type *double* and one *bulk* attribute with "garbage" characters to ensure that each tuple is 100 bytes long. The values of the d *doubles* of a tuple are generated randomly in the range of [0,1). In all experiments, we ask Skyline queries that look for tuples that have low values in all d dimensions. We study three different kinds of databases that differ in the way the values are generated:

indep: for this type of database, all attribute values are generated independently using a uniform distribution. Figure 10 shows such an independent database with 1000 points and $d = 2$. Points which are part of the Skyline are marked as bold circles. In Figure 10, twelve points are part of the Skyline.

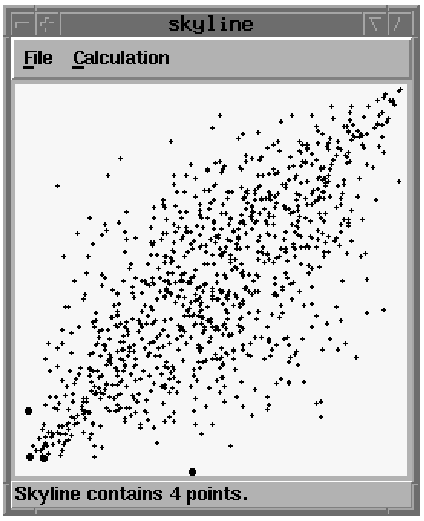
corr: a correlated database represents an environment in which points which are good in one dimension are also good in the other dimensions. For instance, students which have a good publication record typically also do well in their preliminaries. We generate a random point in the correlated database as follows. First we select a plane perpendicular to the line from (0, . . . , 0) to (1, . . . , 1) using a normal distribution; the new point will be in that plane. We use a normal distribution to select the plane so that more points are in the middle than at the ends. Within the plane, the individual attribute values are again generated using a normal distribution; this makes sure that most points are located close to the line from (0, . . . 0) to (1, . . . 1). Figure 11 shows a correlated database with 1000 points for $d = 2$. The Skyline of this database contains four points.

anti: an anti-correlated database represents an environment in which points which are good in one dimension are bad in one or all of the other dimensions; hotels seem to fall into this category. As for a correlated database, we generate a point by first selecting a plane perpendicular to the line from (0, . . . , 0) to (1, . . . , 1) using a normal distribution. We use a normal distribution with very small variance so that all points are placed into planes which are *close* to the plane through the point (0.5, . . . , 0.5). Within the plane, the individual attribute values are generated using a uniform distribution. Figure 12 shows an anti-correlated database with 1000 points for $d = 2$. The Skyline of this database contains 36 points.



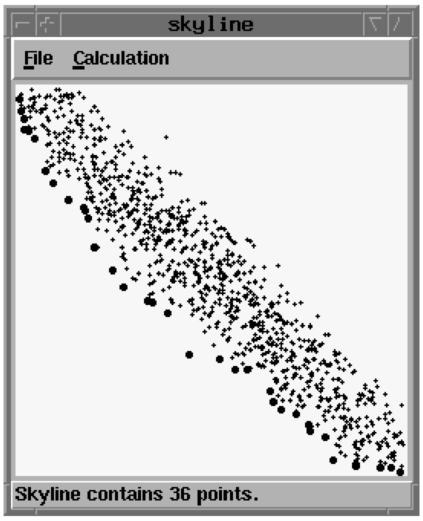
Skyline contains 12 points.

Figure 10: Independent DB



Skyline contains 4 points.

Figure 11: Correlated DB



Skyline contains 36 points.

Figure 12: Anti-Correlated DB

d	*corr*	*indep*	*anti*
2	1	12	49
3	3	69	632
4	11	267	4239
5	17	1032	12615
6	21	1986	26843
7	43	5560	41484
8	121	9662	55691
9	243	16847	67101
10	378	26047	75028

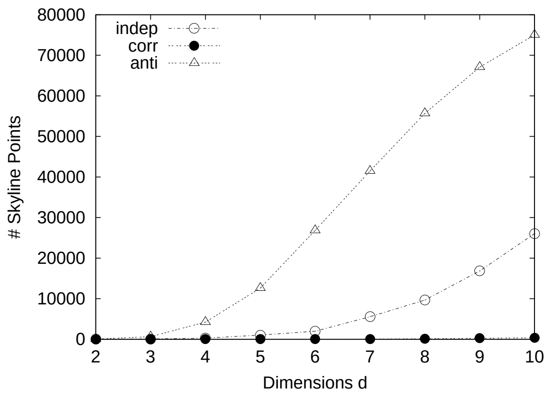


Figure 13: Skyline Sizes

Figure 13 shows the size of the Skyline for each type of database with a growing number of dimensions. These are the databases that we actually used for our experiments; i.e., they all had 100,000 points. While the Skyline is fairly small for all correlated databases, the size of the Skyline increases sharply for the anti-correlated databases. The size of the Skyline of the independent databases is somewhere in between.

5.2 2-d Skylines

Table 1 shows the running times (in seconds) and amount of disk I/O (in MB) for the individual algorithms to compute two-dimensional Skylines for each type of database. We allocated 1 MB of main memory for each of our algorithms; the RDBMS required significantly more. We only show the results for the basic BNL variant here because all three BNL variants have virtually identical performance in this experiment. Furthermore, we could not determine the amount of disk I/O carried out by the RDBMS.

	corr		indep		anti	
	Time	I/O	Time	I/O	Time	I/O
BNL-basic	1.1	10	1.1	10	1.3	10
D&C-basic	88.0	90	87.0	90	89.0	90
D&C-mpt	25.0	30	27.0	30	26.0	30
D&C-mptesk	2.6	10	2.7	10	3.1	10
Sort	2.0	10	1.9	10	2.0	10
RDBMS	28.0	–	37.0	–	92.0	–

Table 1: Running Time (secs) and Disk I/O (MB) of 2-d Skyline Algorithms

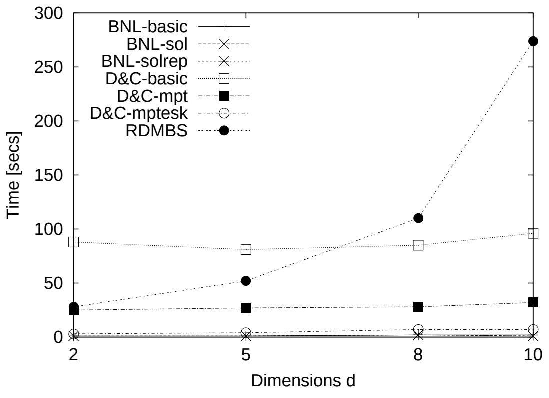


Figure 14: Time (secs), Corr, 1MB Buff.

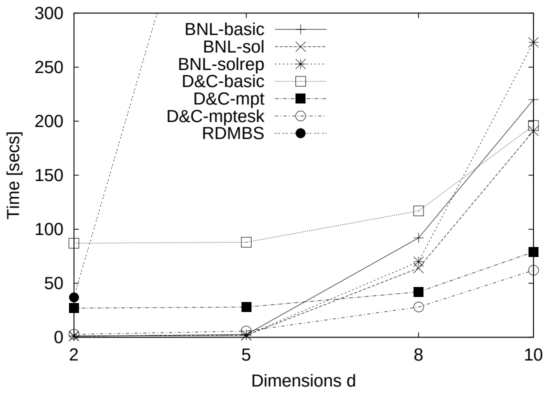


Figure 15: Time (secs), Indep, 1MB Buff.

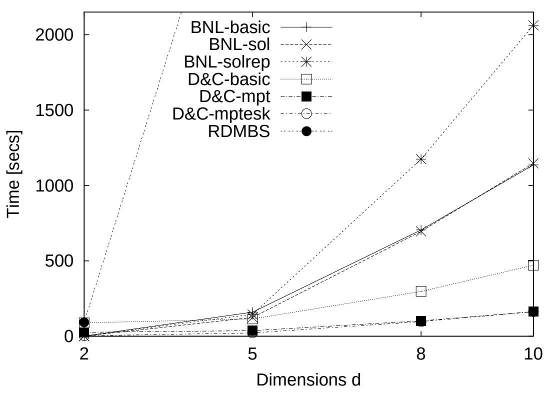


Figure 16: Time (secs), Anti, 1MB Buff.

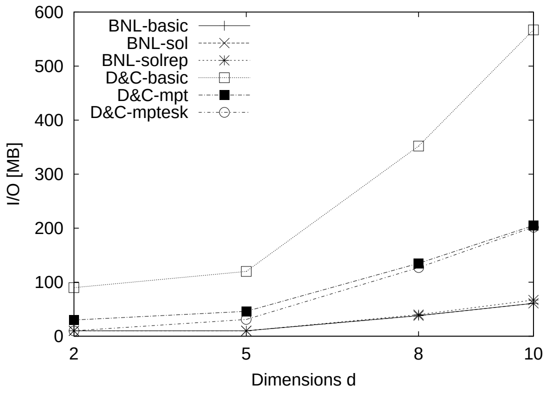


Figure 17: I/O (MB), Anti, 1MB Buff.

The clear winner in this experiment is the BNL algorithm. For all three databases, the window is large enough so that the BNL algorithm terminates after one iteration. The BNL algorithm even outperforms the special Sort algorithm which is only applicable for $d = 2$. The Sort algorithm, however, is also quite fast because an "Early Skyline" is applied before a run is written to disk so that the runs are extremely short. In a different experiment (not shown), we also found a database in which the Sort algorithm actually outperforms the BNL algorithms. In that experiment, we generated a synthetic two-dimensional database with a Skyline of more than 3000 tuples.

"Early Skyline" is also very effective for the D&C algorithm so that the D&C-mptesk is the clear winner of the three D&C variants; after applying an "Early Skyline" the partitions are very small and the rest of the algorithm can be completed very quickly. Applying an "Early Skyline" using Kung et al.'s algorithm, however, is more expensive than sorting or carrying out nested loops in this experiment so that D&C-mptesk is outperformed by the BNL algorithms and Sort. The other D&C variants which do not apply an "Early Skyline" show very poor performance in this experiment because of their high I/O demands; as expected is the basic D&C variant particularly bad.

The RDBMS which carries out a naive nested-loops computation also shows poor performance in this experiment. The RDBMS shows particularly bad performance for the anti-correlated database because in that database on an average more tuples must be probed in the inner loop before a tuple can be eliminated.

5.3 Multi-dimensional Skylines

Figures 14 to 16 show the running times of the alternative algorithms, varying the number of dimensions considered in the Skyline query. The buffer is again limited to 1 MB for our implementation of the algorithms and more for the RDBMS. Turning to Figure 14, we see that the BNL variants are the clear winner for correlated databases. The situation is different for the other databases. For an independent database (Figure 15), the BNL algorithms are good up to five dimensions, after that they are outperformed by the D&C-mpt and D&C-mptesk algorithms. For an anti-correlated database, the break-even point is earlier.

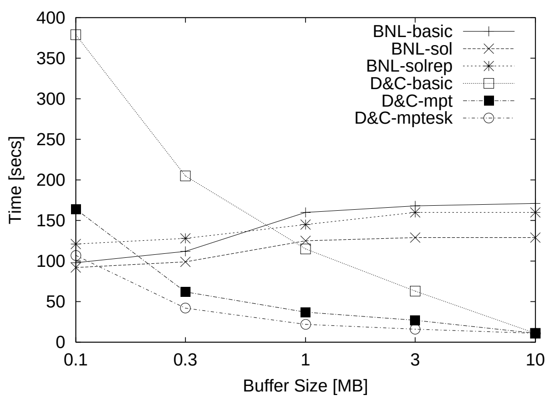


Figure 18: Time (secs), Anti, $d = 5$

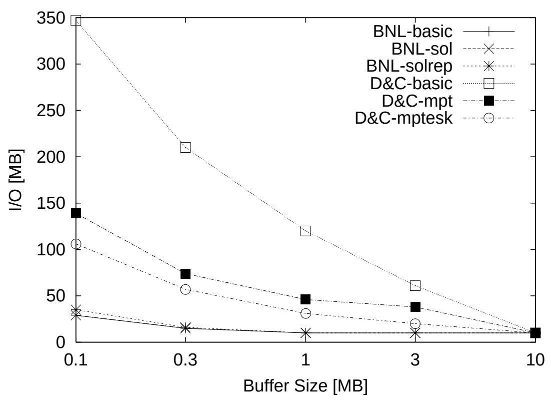


Figure 19: I/O (MB), Anti, $d = 5$

Altogether, we can draw the following conclusions:

- The BNL variants are good if the size of the Skyline is small. As a result, the performance of the BNL algorithms is very sensitive to the number of dimensions and to correlations in the data. Between the three variants, BNL-sol is the overall winner, but the differences are not great. Replacement is clearly a bad idea if the Skyline is very large; in this case, replacement incurs additional overhead without any benefits.

- The D&C variants are less sensitive than the BNL variants to the number of dimensions and correlations in the database. D&C-mptesk is the clear winner among the D&C variants.

- Like the BNL algorithms, the RDBMS is very sensitive to the number of dimensions and correlations in the database. Overall, these experiments confirm that a standard, off-the-shelf database system is not appropriate to carry out Skyline queries. Except for the correlated database, the RDBMS has an unacceptable running time for $d > 2$. It takes, for instance, more than 7.5 hours to compute a ten-dimensional Skyline with an anti-correlated database (mostly CPU time).

In summary, we propose that a system should implement the D&C-mptesk and BNL-sol algorithms.

Figure 17 shows the amount of disk I/O carried out by the alternative algorithms for the anti-correlated database, varying the number of dimensions and with 1 MB of main memory. The trends are the same for the correlated and independent databases. Again, we could not determine the I/O cost of the RDBMS. We see that the BNL algorithms have the lowest I/O cost in all cases; even for ten-dimensional Skylines when their running time is high because of their high CPU overhead. Otherwise, the results are as expected: D&C-mptesk is better than D&C-mpt which in turn is better than D&C-basic. The gap between D&C-mpt and D&C-mptesk narrows with a growing number of dimensions as an “Early Skyline” becomes less and less effective. (The differences are bigger for the other databases.)

5.4 Varying the Size of the Buffer

Figures 18 and 19 show the running times and disk I/O costs of the algorithms to compute a five dimensional Skyline for the anti-correlated database; this time we vary the size of the main-memory buffers from 100 KB (1% the size of the database) to 10 MB (100% the size of the database). We see that the performance of the D&C algorithms improve the larger the memory gets. On the other hand, the running time of the BNL algorithm can get slightly worse with a larger memory. The reason is that the BNL algorithms are CPU-bound so that more memory does not help and that a smaller window results in overall less comparisons. In other words, the BNL algorithms are attractive if memory is scarce and the system is disk-bound.

5.5 Varying the Size of the Database

Table 2 shows the running times of the BNL and D&C variants for a 100 MB anti-correlated database with $d = 5$; i.e., a database with one million points. The size of the Skyline is 35.000 points. The buffer is set

	10 MB Database	100 MB Database
BNL-basic	160	813
BNL-sol	125	605
BNL-solrep	145	924
D&C-basic	115	1768
D&C-mpt	37	388
D&C-mptesk	22	211

Table 2: Running Time (secs); 1 MB Buffer, $d = 5$, *Anti-correlated*, Vary Size of Database

to 1 MB or 1% of the size of the database. As a baseline, Table 2 also shows the results for the 10 MB anti-correlated database which was used in previous experiments. Basically, we observe the same effects for the 100 MB database as for the 10 MB database. In this case, D&C outperforms BNL, and among the D&C variants D&C-mptesk is the clear winner. In fact, m-way partitioning and "Early Skyline" computation become even more important in this case because the larger the database gets, the more critical is it to reduce the I/O complexity.

It is important to note that the size of the Skyline does not grow linearly with the size of the database; in this particular case, the size of the Skyline grew by a factor of three for a database ten times as large. As a result, the BNL variants become more and more attractive the larger the database gets.

We believe that one million points is already a big case for the Skyline operation. We do not expect that the Skyline operation is applied to significantly more points than that. For decision support, for instance, we expect that the Skyline operation will be applied to aggregated data, as in Query 3 of Section 2.

6 Related Work

The closest work to our work is the work by Kung et al. on the maximum vector problem [KLP75]. In that work, the basic divide-and-conquer algorithm was devised and theoretically analyzed. In some sense, our work can be seen as a continuation of that work: we extended their algorithm to work much better in a database environment, we proposed a new block-nested-loops algorithm which is significantly better in good cases, we demonstrated the use of indices, and we showed how the Skyline operator can be combined with other database operations.

Other algorithms for the maximum vector problem have been devised in [SM88, Mat91, RDL95]. These works propose parallel algorithms for the maximum vector problem and/or specific algorithms for cases in which the number of dimensions is 2 or very large. However, none of these approaches have been applied to the database context.

In the "theory" literature a number of related problems have been studied; e.g., the *contour problem* [McL74] or *multi-objective optimization* using linear programming [Ste86]. A more related problem is to compute the *convex hull* of a data set. The convex hull is a subset of the Skyline. The Skyline contains all points which may be optimal for *any* monotonic scoring function; in contrast, the convex hull contains only those points which may be optimal for a *linear* scoring function. Computing the convex hull has been studied intensively in theory (see, e.g., [PS85]), but to the best of our knowledge, this problem has not been addressed in the database context either. Computing the convex hull is a tougher problem (at least in theory). The best known algorithm to compute the convex hull has a complexity of $\mathcal{O}(n^{\lfloor d/2 \rfloor + 1})$ for $d > 3$ and $\mathcal{O}(n * \log n)$ for $d = 2, 3$. Thus for $d > 3$, the complexity of the best algorithm for the convex hull is higher than the complexity of all our algorithms for the Skyline.

In the database area, work on nearest neighbor search and *Top* N query processing is related. For instance, Roussopoulos et al. studied a branch-and-bound algorithm for nearest-neighbor search using R-trees [RKV95]; other algorithms for nearest neighbor search with multi-dimensional search trees have been studied by Berchtold et al. [BBKK97]. As mentioned in Section 4.3 some of the ideas to *push* Skyline operators *through* joins have been adopted from previous work on *Top* N query processing [CK97]. However, nearest neighbor search and *Top* N are different problems. A nearest neighbor search for an *ideal* hotel that costs \$0 and has 0 miles distance to the beach would certainly return some interesting hotels, but it would also miss interesting hotels such as the YMCA which are extremely cheap but far away from the beach.

Furthermore, a nearest neighbor search would return non-interesting hotels which are dominated by other hotels. Likewise a *Top* N query [CK97] would find some interesting hotels, miss others, and also return non-interesting hotels. In practice, we envision that for many applications users will ask for the Skyline first in order to get the "big picture" and then apply a *Top* N query to get more specific results. For applications such as the visualization of the *Skyline* of Manhattan, the Skyline operator is indispensable and nearest neighbor search and *Top* N do not help.

7 Conclusion

We showed how a database system can be extended in order to compute the *Skyline* of a set of points. We proposed the `SKYLINE OF` clause as a simple extension to SQL's `SELECT` statement, presented and experimentally evaluated alternative algorithms to compute the Skyline, discussed how indices can be used to support the Skyline operation, and described how the Skyline operation interacts with other query operators (i.e., *join* and *Top* N). The Skyline operation is useful for a number of database applications, including decision support and visualization. Our experimental results indicated that a database system should implement a block-nested-loops algorithm for good cases and a divide-and-conquer algorithm for tough cases. More specifically, we propose to implement a block-nested loops algorithm with a window that is organized as a self-organizing list and a divide-and-conquery algorithm that carries out m-way partitioning and "Early Skyline" computation.

There are several avenues for future work. One open question is what kind of statistics need to be maintained in order to estimate the size of a Skyline; for example, such an estimate is necessary in order to help the query optimizer decide whether a block-nested-loop or a divide-and-conquer algorithm should be used. Initial work in this direction is reported in [BKST78]. That work, however, assumes that the attribute values are independent which is an unrealistic assumption; for instance, hotels at the beach are typically expensive. Another open question is to find the best algorithms to compute the Skyline with a *parallel* database system and to study the use of indices more closely. Furthermore, we plan to study *approximate* and *online* Skyline algorithms which compute a good approximation very quickly and improve their results the longer they run.

Acknowledgments We would like to thank Christian Böhm, Reinhard Braumandl, and Alfons Kemper for many helpful discussions.

References

[BBKK97] S. Berchtold, C. Böhm, D. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 78–86, Tucson, AZ, USA, May 1997.

[BCK98] R. Braumandl, J. Claussen, and A. Kemper. Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 110–121, New York, USA, August 1998.

[BD83] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Trans. on Database Systems*, 8(2):255–265, 1983.

[BKST78] J. L. Bently, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the avarage number of maxima in a set of vectors and applications. *Journal of the ACM*, 25(4):536–543, 1978.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, AZ, USA, May 1997.

[CK98] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 158–169, New York, USA, August 1998.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 216–226, Montreal, Canada, June 1996.

[GKG+97] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl. Query evaluation in CROQUE – calculus and algebra coincide. In *Proc. British National Conference on Databases (BNCOD)*, pages 84–100, London, UK, July 1997.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HCLS97] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[Lar97] P. A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. Microsoft Technical Report, January 1997. `http://www.research.microsoft.com/~palarson/`.

[Mat91] J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, June 1991.

[McL74] D. H. McLain. Drawing contours from arbitrary data points. *The Computer Journal*, 17(4):318–324, November 1974.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, Berlin, etc., 1985.

[RDL95] C. Rhee, S. K. Dhall, and S. Lakshmivarahan. The minimum weight dominating set problem for permutation graphs is in NC. *Journal of Parallel and Distributed Computing*, 28(2):109–112, August 1995.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 71–79, San Jose, CA, USA, May 1995.

[SM88] I. Stojmenovic and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2):249–251, June 1988.

[Ste86] R. E. Steuer. *Multiple criteria optimization.* Wiley, New York, 1986.

A Block-Nested-Loops Algorithm

$\mathcal{M}$	input of the Skyline operation; a set of d-dimensional points
$\mathcal{R}$	output of the Skyline operation; a set of d-dimensional points
$\mathcal{T}$	temporary file; a set of d-dimensional points
$\mathbb{S}$	main memory; a set of d-dimensional points
$p \prec q$	point p is dominated by point q

```
**function** SkylineBNL($\mathcal{M}$)
**begin**
// Initialization
$\mathcal{R}$ := $\emptyset$, $\mathcal{T}$ := $\emptyset$, $\mathbb{S}$ := $\emptyset$
$CountIn$ := 0, $CountOut$ := 0
// Scanning the database repeatedly
**while** $\neg$EOF($\mathcal{M}$) **do begin**
    **foreach** $p \in \mathbb{S}$ **do**                              // propagate points that have been compared to all
        **if** TimeStamp($p$)=$CountIn$ **then** save($\mathcal{R}$,$p$), release($p$)
    load($\mathcal{M}$,$p$), TimeStamp($p$) := $CountOut$                                      // load next point
    $CountIn$ := $CountIn + 1$
    **foreach** $q \in \mathbb{S} \setminus \{p\}$ **do begin**                       // compare it to all points in memory
        **if** $p \succ q$ **then** release($p$), **break**
        **if** $p \prec q$ **then** release($q$)
    **end**
    **if** $\neg$MemoryAvailable **then begin**                              // write it to tempfile if necessary
        save($\mathcal{T}$,$p$), release($p$)
        $CountOut$ := $CountOut + 1$
    **end**
    if EOF($\mathcal{M}$) then begin                                      // continue with next run if necessary
        $\mathcal{M}$ := $\mathcal{T}$, $\mathcal{T}$ := $\emptyset$
        $CountIn$ := $0$, $CountOut$ := $0$
    end
end
// Flushing the memory
foreach $p \in \mathbb{S}$ do save($\mathcal{R}$,$p$), release($p$)
return $\mathcal{R}$
end
```

B Basic Divide-and-Conquer Algorithm

```
function SkylineBasic($M$,$Dimension$)
begin
if $|M| = 1$ then return $M$                                   // terminate the recursion
$Pivot$ := Median$\{m_{Dimension}|m \in M\}$                    // equi-partition the set
($P_1$,$P_2$) := Partition($M$,$Dimension$,$Pivot$)
$S_1$ := SkylineBasic($P_1$,$Dimension$)                        // compute skyline recursively
$S_2$ := SkylineBasic($P_2$,$Dimension$)
return $S_1 \dot\cup$ MergeBasic($S_1$,$S_2$,$Dimension$)
end

function MergeBasic($S_1$,$S_2$,$Dimension$)
begin
if $S_1 = \{p\}$ then $R$ := $\{q \in S_2|p \not\prec q\}$      // trivial cases
else if $S_2 = \{q\}$ then begin
    $R$ := $S_2$
    foreach $p \in S_1$ do if $p \prec q$ then $R$ := $\emptyset$
end else if $Dimension = 2$ then begin                          // dimension is low
    $Min$ := Minimum$\{p_1|p \in S_1\}$
    $R$ := $\{q \in S_2|q_1 < Min\}$
end else begin                                                  // general case
    $Pivot$ := Median$\{p_{Dimension-1}|p \in S_1\}$ // partition both sets
    ($S_{1,1}$,$S_{1,2}$) := Partition($S_1$,$Dimension - 1$,$Pivot$)
    ($S_{2,1}$,$S_{2,2}$) := Partition($S_2$,$Dimension - 1$,$Pivot$)
    $R_1$ := MergeBasic($S_{1,1}$,$S_{2,1}$,$Dimension$) // compare adjacent parts
    $R_2$ := MergeBasic($S_{1,2}$,$S_{2,2}$,$Dimension$)
    $R_3$ := MergeBasic($S_{1,1}$,$R_2$,$Dimension - 1$) // compare “diagonally”
    $R$ := $R_1 \dot\cup R_3$
end
return $R$
end
```

C M-way Divide-and-Conquer Algorithm

```
**function** SkylineMway($\mathcal{M}$,$Dimension$)
**begin**
$Partitions$ := Minimum$\{2^n | n \in \mathbf{N} \wedge |\mathbb{S}| \cdot 2^n > |\mathcal{M}|\}$    // partition the set
$Quantiles$ := $\alpha$-Quantiles($\mathcal{M}$,$Dimension$,$Partitions$)
($\mathcal{P}_1, ..., \mathcal{P}_{Partitions}$) := Partition($\mathcal{M}$,$Dimension$,$Quantiles$)
**for** $i$ := 1 **to** $Partitions$ **do**    // compute skyline recursively
    **if** $|\mathcal{P}_i| < |\mathbb{S}|$ **then** $\mathcal{S}_i$ := SkylineBasic($\mathcal{P}_i$,$Dimension$)
        **else** $\mathcal{S}_i$ := SkylineMway($\mathcal{P}_i$,$Dimension$)
**while** $Partitions > 1$ **do begin**
    **for** $i$ := 1 **to** $\frac{Partitions}{2}$ **do** $\mathcal{S}_i$ := $\mathcal{S}_i \dot{\cup}$ MergeMway($\mathcal{S}_{2i-1}$,$\mathcal{S}_{2i}$,$Dimension$)
    $Partitions$ := $\frac{Partitions}{2}$
**end**
**return** $\mathcal{S}_1$
**end**

**function** MergeMway($\mathcal{S}_1$,$\mathcal{S}_2$,$Dimension$)
**begin**
**if** $\mathcal{S}_1 = \{p\}$ **then** $\mathcal{R}$ := $\{q \in \mathcal{S}_2 | p \not\prec q\}$    // trivial cases
**else if** $\mathcal{S}_2 = \{q\}$ **then begin**
    $\mathcal{R}$ := $\mathcal{S}_2$
    **foreach** $p \in \mathcal{S}_1$ **do if** $p \prec q$ **then** $\mathcal{R}$ := $\emptyset$
**end else if** $|\mathcal{S}_1| + |\mathcal{S}_2| < |\mathbb{S}|$ **then** $\mathcal{R}$ := MergeBasic($\mathcal{S}_1$,$\mathcal{S}_2$,$Dimension$)
**else if** $Dimension = 2$ **then begin**    // dimension is low
    $Min$ := Minimum$\{p_1 | p \in \mathcal{S}_1\}$
    $\mathcal{R}$ := $\{q \in \mathcal{S}_2 | q_1 < Min\}$
**end else begin**    // general case
    $Partitions$ := Maximum$\{\lceil\frac{|\mathcal{S}_1|}{|\mathbb{S}|/2}\rceil, \lceil\frac{|\mathcal{S}_2|}{|\mathbb{S}|/2}\rceil\}$ // partition both sets
    $Quantiles$ := $\alpha$-Quantiles($\mathcal{S}_1$,$Dimension - 1$,$Partitions$)
    ($\mathcal{S}_{1,1}, ..., \mathcal{S}_{1,Partitions}$) := Partition($\mathcal{S}_1$,$Dimension - 1$,$Quantiles$)
    ($\mathcal{S}_{2,1}, ..., \mathcal{S}_{2,Partitions}$) := Partition($\mathcal{S}_2$,$Dimension - 1$,$Quantiles$)
    $\mathcal{R}$ := $\emptyset$
    **for** $j$ := 1 **to** $Partitions$ **do begin** // compare all parts
        **for** $i$ := 1 **to** $j$ **do**
            **if** $i < j$ **then** $\mathcal{S}_{2,j}$ := MergeMway($\mathcal{S}_{1,i}$,$\mathcal{S}_{2,j}$,$Dimension - 1$)
                **else** $\mathcal{S}_{2,j}$ := MergeMway($\mathcal{S}_{1,i}$,$\mathcal{S}_{2,j}$,$Dimension$)
        append($\mathcal{R}$,$\mathcal{S}_{2,j}$)
    **end**
**end**
**return** $\mathcal{R}$
**end**
```