Efficient Progressive Skyline Computation

Kian-Lee Tan Pin-Kwang Eng Beng Chin Ooi

Department of Computer Science
National University of Singapore

Abstract

In this paper, we focus on the retrieval of a set of interesting answers called the *skyline* from a database. Given a set of points, the skyline comprises the points that are not *dominated* by other points. A point dominates another point if it is *as good or better in all dimensions and better in at least one dimension*. We present two novel algorithms, Bitmap and Index, to compute the skyline of a set of points. Unlike most existing algorithms that require at least one pass over the dataset to return the first interesting point, our algorithms progressively return interesting points as they are identified. Our performance study further shows that the proposed algorithms provide quick initial response time with Index being superior in most cases.

1 Introduction

Database management systems have been increasingly used in decision support applications. Many of these applications are characterized by several features. First, the query is typically based on multiple, and sometimes conflicting, goals. For example, a tourist may be interested in *budget* hotels with *reasonable ratings* (say, 3-star) that are *close to* the city. Clearly, hotels nearer the city are expected to be more expensive. Second, unlike conventional applications, there may be no single *optimal* answer (or answer set). In our tourist example, it is unlikely that there exists a single 3-star hotel that is cheapest among all 3-star hotels and is within the city. Instead, one can expect to find a list of budget hotels such that those nearer to the city are slightly more expensive. Third, because of the second point, users are typically looking for *satisficing* answers. Fourth, for the same query, different users, dictated by their personal preferences, may find different answers appealing. A person may be willing to pay a little more to be closer to the city; another may be contented with a cheaper hotel as long as it is convenient to go to the city. As such, it is important for the DBMS to present *all interesting* answers that may fulfill a user's need.

Traditionally, the DBMS supports these applications by returning all answers that may meet the user's requirement. In our tourist example, if the user specifies "budget" to mean in the range of \$120-\$200, and "close to" to mean within 5 km, then the system may return all hotels that satisfy these predicates. This is not very helpful because users may be overloaded with too much information. More importantly, there may be answers that are completely irrelevant and uninteresting. For example, if there are two hotels, h1 and h2, with the same rating, such that h1 is both cheaper and nearer to the city than h2. Then, h2 may not need to be presented to the user.

In this paper, we focus on the set of interesting answers called the *skyline* [2]. Given a set of points, the skyline comprises the points that are not *dominated* by other points. A point dominates another point if it is *as good or better in all dimensions and better in at least one dimension*. In our example, h1 dominates h2 because it is better in the price and distance dimensions and as good in the rating dimension. We present two novel algorithms to compute the skyline of a set of points. Unlike most existing algorithms [2] that require at least one pass over the dataset to return the first interesting point, our algorithms progressively return the interesting points as they are identified.

The first algorithm, called Bitmap, is *completely* non-blocking and exploits a bitmap structure to quickly identify whether a point is an interesting point or not. Each record is mapped into a m-bit vector, where m is the sum of the number of distinct attribute values over all dimensions. Unlike existing bitmap structures which are typically a bitmap version of the entire database, our bitmap structure is a precomputed bit structure with more information. Operations on the bit vectors are performed on the

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

bit-slices derived from the vectors. Because bitwise operations are fast, and the precomputed information bounds the number of bit-slices that need to be examined, we can efficiently determine whether a point is an interesting point or not.

The second method, Index, exploits a transformation mechanism and a B^+-tree index to return skyline points in batches. Essentially, each point is transformed into a single dimensional space, and stored in a B^+-tree structure. Moreover, points with some common features (same mapping value) are clustered together. The sort order in the transformed space allows us to examine points that are likely candidates to be skyline points first. Moreover, it also allows us to prune away points that are clearly dominated by some other points. By processing points with the common features collectively, we can determine the skyline points in bursts.

We implemented the proposed algorithms and evaluated their performance against three recently proposed algorithms. Our results show that the proposed schemes provide short initial response time and return interesting answers very quickly. Moreover, both schemes can also outperform the existing techniques in terms of total response time. While Index is superior in most cases, Bitmap performs well when the number of distinct values per dimension is small.

The rest of this paper is organized as follows. In the next section, we review the skyline operator and existing algorithms that compute the skyline. In Section 3, we present the bitmap-based and index-based approaches to support progressive computation of skyline. Section 4 reports a performance study that evaluates the proposed schemes against existing algorithms, and finally, we conclude in Section 5.

2 The Skyline Operator

In [2], Borzsonyi et. al. extended SQL's SELECT statement by an optional `SKYLINE OF` clause. The `SKYLINE OF` clause is evaluated after the `SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...` part of the query, but before the `ORDER BY` clause (and `STOP AFTER` if supported). The SKYLINE OF clause selects all interesting records, i.e., records that are not dominated by other records. Formally, a tuple $p = (p_1, \ldots, p_k, p_{k+1}, \ldots, p_l, p_{l+1}, \ldots, p_m, p_{m+1}, \ldots, p_n)$ dominates another tuple $q = (q_1, \ldots, q_k, q_{k+1}, \ldots, q_l, q_{l+1}, \ldots, q_m, q_{m+1}, \ldots, q_n)$ for a Skyline query

$$\begin{array}{ll} \texttt{SKYLINE OF} & d_1\texttt{MIN}, \ldots, d_k\texttt{MIN}, \\ & d_{k+1}\texttt{MAX}, \ldots, d_l\texttt{MAX}, \\ & d_{l+1}\texttt{DIFF}, \ldots, d_m\texttt{DIFF} \end{array}$$

if the following three conditions hold:

$$\begin{array}{ll} p_i \leq q_i & \text{for all } i = 1, \ldots, k \\ p_i \geq q_i & \text{for all } i = k+1, \ldots, l \\ p_i = q_i & \text{for all } i = l+1, \ldots, m \end{array}$$

In the SKYLINE OF clause, the `MIN` and `MAX` annotations mean that the corresponding dimensions should be minimized and maximized respectively, and the `DIFF` annotation denotes that two records with different values in the dimension may both be part of the skyline.

In our tourist example where both the price and distance are to be minimized, the query can be written in SQL as follows:

```
SELECT *
FROM hotels
WHERE rating > 2
SKYLINE OF price MIN, distance MIN;
```

where `hotels(hotelid, address, typeOfRoom, rating, price, distance)` is a relation on hotel information, `rating` represents the rating of the hotel (here, we assume its type is integer, and the value represents the number of stars), `price` captures the room rates, and `distance` indicates the distance from the hotel to the city. Should the user wants to maximize the value of an attribute (e.g., one may want to maximize the rating), then the `MAX` annotation can be used. Similarly, the user may use the `DIFF` annotation to indicate that two records with different room types are acceptable.

Computing the skyline of a set of points is also known as the maximum vector problem [1, 6]. Early works on solving the maximum vector problem typically assume that the points fit into the main memory. Algorithms devised include divide-and-conquer paradigm [1], parallel algorithms [8, 9] and those that are specifically designed to target at 2 or very large number of dimensions [4]. For the rest of this section, we shall review three techniques that were proposed in the context of database applications in [2]. These are the three schemes that we shall compare with in our performance study.

Block Nested Loops Algorithm

The block nested loops algorithm [2] is an iterative algorithm that repeatedly scans a set of records. In each iteration, a *window* of incomparable records are kept in the main memory. When a record p is read from the input relation, p is compared with the records in the window. There are three possible outcomes:

1. If p is dominated by a record in the window, it means that p cannot be in the skyline. Thus, p is discarded.

2. If p dominates one or more records in the window, these records are eliminated (since they cannot be in the skyline), and p is inserted into the window.

3. If p is incomparable with all records in the window (i.e., it neither dominates nor being dominated), it is either inserted into the window if there is sufficient room in the window, or written to a temporary file on disk.

At the end of each iteration, those records in the window that have been compared against all records that have been written out to the temporary file are certain to be in the skyline, and hence can be returned to the user (and removed from the window). In the next iteration, the algorithm will proceed in the same manner with the remaining records in the window and the records in the temporary file as the input relation.

One of the most expensive process in the algorithm is the time spent to compare a record with the records in the window. To speed up these comparisons, the records in the window are organized as a *self-organizing list*. When a record q of the window is found to dominate another record, then q is moved to the beginning of the window. In this way, records that are highly dominant will float to the top of the window, and subsequent input records will be compared against them first.

The algorithm clearly is not attractive in terms of producing fast initial response time as it requires at least one pass over the input relation before a set of points can be identified to be part of the skyline.

Divide-and-Conquer Algorithm

In [2], Borzsonyi et. al. also extended the basic two-way divide-and-conquer algorithm for computing skyline [1, 6] to m-way partitioning. The algorithm works as follows.

- Compute the α-quantiles of a set of input points along a certain dimension d_p. Split the points into m partitions such that each partition fits in the memory. Let these partitions be $P_1, \ldots, P_m$.

- Compute the skyline S_i of partition P_i, $1 \leq i \leq m$, using any known skyline computation algorithm.

- Compute the overall skyline as the result of merging the S_i pairwise. Within the merge operation, m-way partitioning is applied so that all sub-partitions can be merged in main memory. We note that the recursive applications of the m-way partitioning pick different dimensions for splitting. It is also interesting to note that some pairs of merging can be skipped as points within these pairs are already incomparable.

Like the nested-loops scheme, the divide-and-conquer technique cannot produce skyline points progressively. Moreover, it is not expected to perform well for small memory systems, as it requires the partitions to be in-memory.

Using B-trees

The computation of the skyline can also be facilitated by index structures. In [2], a method based on B-tree was described. Assuming that each record has d dimensions and there is an index for every dimension, the skyline can be computed as follows.

- Scan all the indexes *simultaneously* to find the first match, i.e., the first record to be seen by all the indexes during the scan.

- The first match is definitely part of the skyline and can be returned immediately, providing a fast initial response.

- Scan the rest of the index entries of the first dimension's index. If the record has not been seen before (i.e., the index entries of this record in the other indexes have not been examined prior to the first match), it is definitely not in the skyline and can thus be eliminated. If any of the other indexes contain an index entry to this record prior to the first match, then the record may or may not be in the skyline. To determine whether it is in the skyline, an existing skyline computation algorithm can be applied.

A critical factor that will affect the performance of this algorithm is how fast the first match can be found. If a match is found late (which is likely to be the case for large number of dimensions), it will result in a high initial response time. Nevertheless, we can expect this algorithm to perform well in general, when the skyline is small and the first match can be found quickly.

3 Progressive Skyline Computation

In this section, we shall present the two proposed methods to compute skyline progressively. For pedagogical reasons, we shall assume that the database, D, contains $|D|$ d-dimensional points. Moreover, we assume that the skyline operation involves all the d dimensions, and that dimension i has k_i distinct values, $1 \leq i \leq d$. Let p_{ij} denote the jth distinct value of the ith dimension. Without loss of generality, we assume that $p_{i1} > p_{i2} > \ldots > p_{ik_i}$. In addition, we assume that the domain for all dimensions are the same and is in the range [0,1]. In presenting the proposed schemes, we shall also restrict our discussion to the `MAX` annotations only. In [10], we discuss how the schemes can be easily generalized to handle skyline queries involving fewer than d dimensions and other annotations (i.e., `MIN` and `DIFF`), as well as involving databases whose dimensions' domains are different.

3.1 Bitmap: A Bitmap-based Algorithm

To support progressive skyline computation, for each point examined, the Bitmap scheme asks the question: "Is this point dominated by another point?". The point is an interesting point if the answer is negative, and we can return it immediately. As such, the method is completely non-blocking, and the initial response time is short compared to existing schemes. Intuitively, to realize this, we need to examine all points in the database. We avoid this by exploiting a bitmap structure. Since bitwise operations are fast, we can efficiently determine whether a point is an interesting point or not.

A point $x = (x_1, \ldots, x_d)$ in the database is represented as a m-bit vector as follows:

- x_i is represented by k_i bits. Hence $m = \sum_{j=1}^{d} k_j$.
- Let the jth bit correspond to p_{ij}. Note that since we are considering the MAX annotation, the first bit corresponds to p_{i1} (which corresponds to the largest value in the dimension), the second bit corresponds to p_{i2} (which corresponds to the second largest value), and so on. If x_i is the p_{iq}th distinct value of dimension i, then the k_i bits representing x_i are set as follows: bits 1 to $q-1$ are set to 0, and bits q to k_i are set to 1.

Figure 1 shows an example. Here, we have a table containing four 3 dimensional data points (Figure 1a). The first dimension has 4 distinct values (4, 3, 2, 1), the second dimension has 3 distinct values (3, 2, 1) and the third dimension has 2 distinct values (2, 1). Consider the second tuple `(3 2 1)`. The value in its first dimension is 3, which is the second largest value. So, only the bit corresponding to 4 is set to 0, while the other bits are set to 1, resulting in the sequence `0111`. Similarly, for the second dimension, its value is 2, and so only the bits corresponding to values larger than 2 (in this case, only one of them which has value 3) will be set to 0, while the rest are set to 1. This leads to the sequence `011` in the second dimension for this tuple. Finally, using the same logic, the bit sequence corresponding to the third dimension of the tuple is `01`. We note that the key property of this bitmap is that by looking at a bit of a vector, it tells us the values that a tuple is as good as or better than. For example, by looking at the bit sequence of tuple 2's first dimension (i.e., `0111`), bit 3 tells us that tuple 2's first dimension is as good as or better than 2.

For efficient computation, the array of vectors obtained from all points are transposed into an array of *bit-slices*, and the file is stored as slices.

Let BS_{ij} denote the bit-slice for the jth distinct value of the ith dimension. Now, we discuss how we determine whether a point $x = (x_1, x_2, \ldots, x_d)$ is in the skyline:

d1	d2	d3
1	1	2
3	2	1
4	1	1
2	3	2

(a) Data points

d1				d2			d3	
4	3	2	1	3	2	1	2	1
0	0	0	1	0	0	1	1	1
0	1	1	1	0	1	1	0	1
1	1	1	1	0	0	1	0	1
0	0	1	1	1	1	1	1	1

(b) The corresponding bitmap structure

Figure 1: An example to illustrate the bitmap-based method.

1. Let $A = BS_{1q_1}$ & BS_{2q_2} & $\ldots$ & BS_{dq_d}
   where & represents the bitwise *and* operation, and x_i is the q_ith distinct value at dimension i. The result of this operation, bit-slice A, has the property that the nth bit is set to 1 if and only if the nth point has value in each dimension greater or equal to the value of the corresponding dimension in x.
2. Let $B = BS_{1q_1-1} \mid BS_{2q_2-1} \mid \ldots \mid BS_{dq_d-1}$
   where $\mid$ represents the bitwise *or* operation. In this step, we take the preceding bit-slice of BS_{iq_i} (which is corresponds to the smallest value that is larger than x_i) and *or* them. If there is no preceding bit-slice for dimension i (i.e., when q_i corresponds to the first bit-slice), then the bit-slice BS_{i0} is set to 0. The result of this operation, bit-slice B, has the property that the nth bit is set to 1 if and only if the nth point has some of its dimension's value greater than the value of the corresponding dimension in x.
3. Let $C = A$ & B
   The result of the operation, C, has the property that the nth bit is set to 1 if and only if the nth point has each dimension's value greater or equal to the corresponding dimension's value in x **and** some of its dimension's value is strictly greater than the corresponding dimension's value in x. Hence, we can conclude that the nth point dominates x. Conversely, if the resultant bit-slice has a value of zero, it tells us that there is NO such point in the database that dominates x and we can conclude that x is a skyline point.

Referring to our example, we now illustrate how to determine whether the second point (3, 2, 1) is in the skyline or not. First, we carry out the bitwise *and* operation:

$$A = 0110 \,\&\, 0101 \,\&\, 1111 = 0100$$

Next, we carry out the bitwise *or* operation:

$$B = 0010 \mid 0001 \mid 1001 = 1011$$

Finally, we carry out the bitwise *and* operation:

$$C = 0100 \,\&\, 1011 = 0000$$

Since the answer is zero, no points in the database dominates (3, 2, 1). Thus, (3, 2, 1) is a skyline point. This example clearly shows that as each point is examined, we can determine easily whether it is a skyline point!

Figure 2 gives an algorithmic description of the proposed Bitmap technique. Routine **BitSlice**(q_i, i) retrieves the bit-slice for the q_ith distinct value of the ith dimension. If the bit-slice does not exist, then **BitSlice**(q_i, i) returns 0. The algorithm starts by looping through each point x in the database. For each point in the database, the bit-slices for each dimension's value of x are first retrieved and bitwise *and* together (lines 3-5). Next, the preceding bit-slices for each dimension's value of x are retrieved if they exist. Subsequently, they are bitwise *or* (lines 6-8) together. Finally, a bitwise *and* operation is applied on the two resultant bit-slices and the results are checked (lines 9-11). If the result is zero, we can conclude that no points in the database dominates x, i.e., x is an interesting point and we output x.

Algorithm Bitmap

1. for each point $x = (x_1, x_2, \ldots, x_d)$ in the database
2. let x_i be the q_ith distinct value in dimension i
3. $A \leftarrow BitSlice(q_1, 1)$
4. for $i = 2$ to d
5. $A \leftarrow A$ & $BitSlice(q_i, i)$
6. $B \leftarrow BitSlice(q_1 - 1, 1)$
7. for $i = 2$ to d
8. $B \leftarrow B \mid BitSlice(q_i - 1, i)$
9. $C \leftarrow A$ & B
10. if $C == 0$
11. output x

Figure 2: Bitmap-based algorithm.

3.2 Index: A B$^+$-tree-based Algorithm

The Index scheme exploits a transformation mechanism that maps high dimensional points into single dimensional space and a B$^+$-tree structure to index the transformed points. The scheme works as follows. Let $(x_1, x_2, \ldots, x_d)$ be an arbitrary point. Recall that we have assumed that $0 \leq x_j \leq 1, 1 \leq j \leq d$. Let x_{max} be largest values among all the d dimensions of the data point. Let the corresponding dimension for x_{max} be d_{max}. The data point is mapped to y over a single dimensional space as follows

$$y = d_{max} + x_{max} \quad (1)$$

We note that the transformation actually organized the data space into different partitions based on the dimension which has the largest value, and provides an ordering within each partition. After the transformation, any single dimensional indexing structure can be used to index the transformed values. In this paper, we adopt the B$^+$-tree structure [3]. However, we assume that the B$^+$-tree leaf nodes are linked to both the left and right siblings [7]. Moreover, we assume that the high dimensional point is kept at the leaf nodes of the tree.

We note that the transformation in Equation 1 is a special case of the more general transformation function used in iMinMax(θ) [5]. This makes the proposed approach more attractive. First, B$^+$-tree is readily available in all existing commercial database systems. Second, we are not advocating a transformation function that is specially tailored to skyline queries. Instead, as shown in [5, 11], similar transformation function can be used to support range and nearest neighbour queries. This means that we only need one index structure to support all three types of queries! Before we present the algorithm for progressive skyline computation, we shall illustrate the idea with an example.

Example 1 *Consider the example shown in Figure 3. Here, we only show the content of the partitions after the transformation (without showing the tree structure). Note that in the figure, each partition is sorted in non-ascending order of the maximum value in that dimension. This can be interpreted as scanning the partition from the last leaf node (and backward) within each partition. We note that we will need an additional pointer to the data record if there are other dimensions that are not indexed (i.e., not used in any skyline operation).*

□

dimension 1	dimension 2	dimension 3
(0.9, 0.8, 0.6)	(0.7, 0.8, 0.5)	(0.1, 0.5, 0.9)
(0.9, 0.5, 0.7)	(0.5, 0.8, 0.6)	(0.8, 0.8, 0.9)
(0.9, 0.2, 0.1)	(0.6, 0.6, 0.6)	(0.7, 0.6, 0.9)
(0.8, 0.7, 0.7)	(0.5, 0.6, 0.6)	(0.2, 0.1, 0.9)
⋮	⋮	⋮
(0.2, 0.2, 0.2)	(0.3, 0.4, 0.2)	(0.3, 0.4, 0.5)
(0.2, 0.1, 0.1)	(0.2, 0.4, 0.1)	(0.2, 0.1, 0.3)
(0.1, 1.1, 0.1)	(0.1, 0.3, 0.2)	(0.2, 0.2, 0.3)

Figure 3: An example for the index-based method.

We made the following interesting (and important) observations. First, we note that the interesting (and potentially dominant) points are largely at the top. In fact, we can identify some interesting points by simply looking at tuples with the largest values in each dimension. In our example, there are 7 tuples (3 from dimension 1, and 4 from dimension 3) that have the maximum value of 0.9 in some dimensions. Among them, it is clear that (0.8,0.8,0.9) in dimension 3 is in the skyline. This means that we can provide very fast initial response time to the user!

Second, we can prune away some of the points easily without examining them. This follows from the fact that if the minimum value among all dimensions in a tuple is larger than the maximum value among all dimensions in another tuple, then the first tuple dominates the second. Clearly, the larger the minimum value is, the more records we can prune. In our example, clearly (0.8,0.8,0.9) dominates all tuples whose maximum value is smaller than 0.8. So, all such tuples need not be examined. Since the structure is organized in sorted order based on the maximum value, this means that we do not need to examine the records to remove them. This translates to saving in I/O cost, and is in contrast with existing algorithms that require the entire relation to be scanned at least once.

Third, in the worst case, we can apply existing techniques by scanning the leaf nodes. Even with this strategy, we can expect a gain over existing scheme, since only the dimensions are involved. Fourth, we can clearly optimize the internal structure by ordering the points with the same maximum value by the minimum values. Fifth, unlike sort-based algorithm which may require large main memory (as dominating points can be far apart), the proposed scheme (as noted in the first point) will not suffer the same problem. As such, we expect the scheme to perform well even with a small amount of memory.

Some of the above observations can be summarized in the following results (Readers are referred to [10] for the proofs of these results). Theorem 1 says that some records can be pruned. Theorem 2 says that skyline points can be obtained from points with the largest value. Theorem 3 further shows that if we were to examine records in descending order of the largest values, then we can find skyline points (from these records) without considering those with smaller values. Note that it is possible to have multiple records with the same transformed value, and so, the collection of records should be considered collectively when determining the skyline points.

Theorem 1 *Consider two points* $x = (x_1, x_2, \ldots, x_d)$ *and* $y = (y_1, y_2, \ldots, y_d)$. *Let* $x_{max} = \max_{i=1}^{d}(x_i)$, $x_{min} = \min_{i=1}^{d}(x_i)$, *and* $y_{max} = \max_{i=1}^{d}(y_i)$. *Let* x_{min} *occurs at dimension* d_{min}, *and* y_{max} *occurs at dimension* d_{max}. *Then, if* $x_{min} > y_{max}$, x *dominates* y.

Theorem 2 *Let* D *be a database containing* $|D|$ d*-dimensional points. We define* m *as*

$$m = \max_{i=1}^{|D|}(\max_{j=1}^{d} x_{ij})$$

where x_{ij} *corresponds to the value of the* j*th dimension of the* i*th point. We define* M *as follows*

$$M = \{(x_1, x_2, \ldots, x_d) | (x_1, x_2, \ldots, x_d) \in D \wedge \max_{i=1}^{d} x_i = m\}$$

Let S_D *be the skyline of* D, *and* S_M *be the skyline of* M. *Then,* $S_M \subseteq S_D$.

Theorem 3 *Let* D *be a database containing* $|D|$ d*-dimensional points. Let there be* k *distinct values in the dimensions of the points in* D. *Let* m_1 *denote the maximum value,* m_2 *denote the second largest value, and so on, and finally,* m_k *denote the minimum value. Moreover, let us split the database* D *into* k *partitions,* $P_1, \ldots, P_k$, *such that*

$$P_i = \{(x_1, x_2, \ldots, x_d) | (x_1, x_2, \ldots, x_d) \in D \wedge \max_{j=1}^{d} x_j = m_i\}$$

Let S_D *be the skyline of* D, *and* S_i *be the skyline of* P_i. *Let us compute* S_D *by examining partitions in the order* P_1, $P_2 \ldots, P_k$. *Then, when we are examining* P_j, *we can determine whether points in* S_j *are in* S_D *without having to look at* $P_{j+1}, \ldots, P_k$.

We are now ready to look at the algorithm. Figure 4 shows the algorithmic description of the proposed index-based scheme. The algorithm is highly abstracted. We shall briefly discuss the routine and variables. f_i is a flag that indicates whether dimension

Algorithm Index

```
1.  for i = 1 to d
2.      f_i ← True
3.      t_i ← traverseTreeMax(root, i)
4.      max_i ← maxValue(t_i)
5.      min_i ← minValue(t_i)
6.  mn ← max_{i=1}^{d} min_i
7.  mx ← max_{i=1}^{d} max_i
8.  for i = 1 to d
9.      if mn > max_i
10.         f_i ← False
11. j ← 1
12. S ← ∅
13. while there are some partitions to be searched
14.     for i = 1 to d
15.         if max_i == mx
16.             P_j ← t_i
17.             S_j ← ∅
18.             t_i ← getNextLeftElement(t_i)
19.             while (maxValue(t_i) == mx)
20.                 mn ← max(mn, minValue(t_i))
21.                 P_j ← P_j ∪ t_i
22.                 t_i ← getNextLeftElement(t_i)
23.             max_i ← maxValue(t_i)
24.     S_j ←computePartitionSkyline(P_j)
25.     S ← S∪ computeNewSkyline(S_j, S)
26.     j ← j + 1
27.     mx ← max_{i=1}^{d} max_i
28.     for i = 1 to d
29.         if mn > max_i
30.             f_i ← False
```

Figure 4: Index-based skyline computation algorithm.

i still needs to be searched. When f_i is set to False, it means that all subsequent records are dominated by some point, and so, partition i need not be searched any further. Routine **maxValue(t)** returns the maximum value among all dimensions of the tuple t. Similarly, **minValue(t)** returns the minimum value among all dimensions of t. **traverseTreeMax(root,i)** is a routine that traverses the B$^+$-tree to obtain the tuple with the largest value in dimension i. Routine **getNextLeftElement**(t) returns the left element of t (if the element is in the left sibling node, then the sibling node will have to be accessed first). Routine **computePartitionSkyline**(P) computes the skyline for a set of points P. Any existing algorithms [2] can be used for the computation. In our implementation, we use the block nested loop algorithm. Routine **computeNewSkyline**(S_j, $\mathcal{S}$) computes the new skyline points to be obtained from S_j and the current skyline points $\mathcal{S}$. Note that the routine also returns these points to the user. It may also involve accessing the data records if not all dimensions are stored at the leaf nodes of the tree.

Steps 1-5 basically begins the search at the last element of each partition. In step 6, we essentially identify the threshold whereby records are guaranteed to be dominated. This is given by mn, the maximum value among all the minimum values in the dimensions of all seen records. Step 7 provides the value (stored in mx) to identify the group of points whose maximum value among all dimensions must take on. Steps 8-10 do the first pruning to eliminate any partitions that need not be searched. Steps 11-30 proceed on to locate any skyline points as follows. Essentially, while there are more partitions to be searched (i.e., some partitions' f_i value is True), the search continues by picking the points that have the current maximum value equals to mx and storing them in a separate partition P_j (steps 14-23). At the same time, mn is updated to reflect the maximum value among all the minimum values of the points examined so far. A higher value will result in fewer partitions that need to be searched subsequently. Next, the skyline of the points in the new partition, P_j is determined (step 24). This new set of skyline points are then compared with the skyline points found so far because some of these new skyline points may be dominated by the current list of skyline points (step 25). Finally, the threshold is updated and more dimensions may be eliminated as a result (steps 26-30). The process repeats itself by looking for the next group of points to examine.

4 A Performance Study

To evaluate the effectiveness of our proposed skyline algorithms, we conducted an extensive set of experiments. This section reports the experimental setup and some representative results. For a more complete set of experiments and results, please see [10].

4.1 Experimental Setup

All the experiments are carried out on a Pentium III PC with a 866 MHz processor and 128 MB of main memory running the Linux operating system. All algorithms are implemented in C or C++.

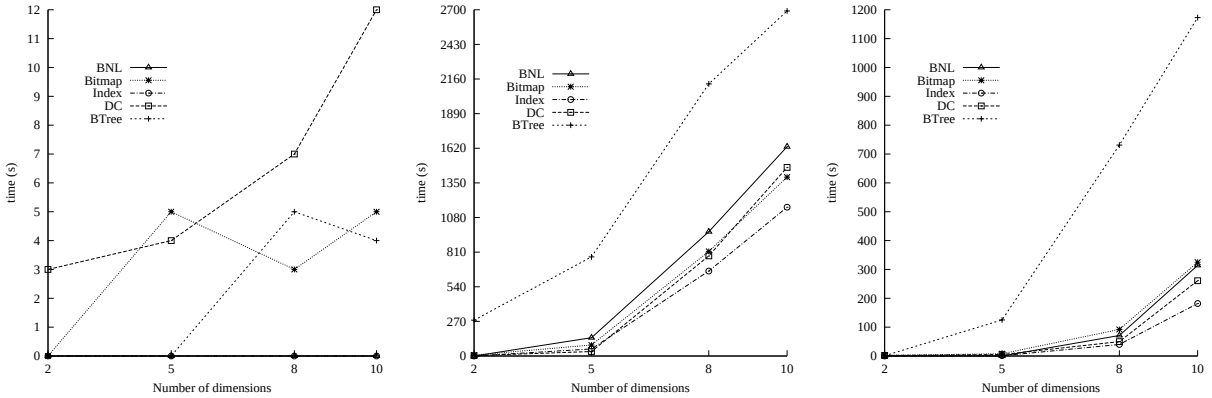
Generating the databases

The databases used in all our experiments are generated in a similar way as described in [2]. Each database contains 100000 tuples, each of size 100 bytes. Each tuple has d dimensions and one "bulk" attribute that is packed with garbage characters to ensure the tuple is 100 bytes long. However, we differ from [2] in that we use integers instead of doubles. We modified the generator used in [2] to generate integers in the range of [1, 100] for our experiments. Three types of databases are generated: (1) **Independent** where the attribute values of the tuples are generated using an uniform distribution; (2) **Correlated** which contain tuples whose attribute values are good in one dimension and are also good in other dimensions; (3) **Anti-correlated** which contain tuples whose attribute values are good in one dimension but are bad in one or all of the other dimensions.

Figure 5 shows the sizes of the skylines for different types of databases using different dimensions. From the figure, we observe that the number of skyline points increases as the number of dimensions increases. It is interesting to note that these values are similar to [2] despite the fact that we are using only distinct integer values for the dimensions. An exception is for a 2 dimensional correlated database where a higher number of equivalent skyline points happened to be generated. Furthermore, we can also see that the size of the skyline for correlated databases is fairly small while it is fairly large for anti-correlated databases, with independent databases somewhere in between. For simplicity, all the skyline queries used in our experiments look for tuples that have high values in all d dimensions i.e., the `MAX` annotation.

Dimension	Correlated	Independent	Anti-Correlated
2	17	9	35
3	3	15	397
4	6	127	2790
5	9	347	10240
6	36	1328	22716
7	61	2831	38117
8	101	7918	51719
9	185	13223	63782
10	215	22367	73200

Figure 5: Skyline sizes



(a) Correlated (b) Anti-correlated (c) Independent

Figure 6: Actual runtime.

4.2 Experimental Results

We describe the results of our experiments in this subsection. We implemented the proposed Bitmap scheme (denoted Bitmap) and the Index scheme (denoted Index). In the current implementation, we enhanced both the schemes by keeping as many of the skyline points found during processing in memory as possible. Hence, before the bitmap or index is used, the input tuple is checked against the current list of skyline tuples and will be eliminated if any of these skyline tuples dominates it. As comparisons, we also implemented the three algorithms proposed in [2]: the block nested loop algorithm where the window is organized as a self-organizing list (denoted BNL), the M-way divide and conquer algorithm (denoted DC), and the B-tree-based scheme (denoted BTree). For BTree, we employ the block nested loop algorithm for cases where it is not possible to determine a skyline point solely through the index.

4.2.1 Experiment 1: Comparing the overall runtime performance

In this experiment, we examine the total amount of time needed by each algorithm to find the skyline. We compare the time taken for each type of database using tuples of dimensions 2, 5, 8 and 10 while maintaining 1 MB of main memory throughout the experiments. Figure 6 shows the results of the experiment.

Figure 6a shows the runtime performance for correlated databases. We observed that both BNL and Index perform better than the rest of the algorithms. This is because the number of skyline points in correlated databases is small. As a result, most of the skyline points can fit into the window in BNL, helping to eliminate many subsequent tuples. For Index, a small skyline results in fewer leaf nodes to be scanned, reducing the time taken significantly. On the other hand, DC and Bitmap are not favorable because the overheads arising from doing the merging in DC and loading the bitmaps in Bitmap are significant compared to the processing time. For BTree, it is only good when the number of dimensions is small because the first match can be determined quickly in low dimensions.

A different scenario arises for anti-correlated databases (Figure 6b). BNL now performs badly for high dimensions (> 5). This result is consistent with the study done in [2]. On the other hand, our Index scheme not only performs well throughout, but is able to outperform BNL by a wide margin when the skyline size is large. This is because most of the time, the indexes are sufficient to eliminate a large number of tuples without retrieving the actual tuples, thereby reducing a substantial amount of runtime. As for BTree, its results is bad throughout. This is inevitable as the attribute values of all dimensions in an anti-correlated database are fairly far apart, thus incurring a high search cost for the first match.

Figure 6c shows the runtime performance for independent databases of various dimensions. From the figure, we can see that Index remains the best while Bitmap's performance decreases because it has to access the bitmaps more frequently as fewer skyline points are found and kept in memory. The performance of the rest of the algorithms remain relatively unchanged compared to using anti-correlated databases except that they took shorter time due to a smaller number of skyline points.

From the results, we can draw the following conclusions. First, our Index scheme is superior than the rest of the algorithms in terms of overall runtime. Second, the Bitmap scheme performs well when the number of dimensions is small.

4.2.2 Experiment 2: Comparing percentage of answers returned at intervals

In this experiment, we examine the performance of the algorithms in terms of how fast answers are returned progressively. Like the previous experiment, we tested the algorithms using different types of databases and

varying the number of dimensions used while maintaining a buffer size of 1 MB. However, besides keeping track of the overall runtime, we also recorded the time taken for each algorithm to output 20%, 40%, 60%, 80% and 100% of the answers. For brevity, we only show the results for the different types of databases at dimension 5 (Figure 7).

Figure 7a shows the results for anti-correlated databases. From the results, several observations can be made. First, both Bitmap and Index can produce tuples much faster than the other algorithms. In fact, the first tuple from Bitmap and Index is almost instantaneous! This clearly illustrates that both our schemes can progressively compute skyline points much faster than the other algorithms. In particular, our Index scheme is the most efficient compared to the rest. Second, DC remains constant for all dimensions because it can only start producing tuples when it completes its execution. BNL, on the other hand, can start producing tuples after the first iteration when all tuples in the database have been examined. Third, although BTree can produce the first tuple much faster than BNL for high dimensions, it is still slow compared to Bitmap and Index. Furthermore, its performance degrades rapidly, making it an undesirable option for progressive computation.

For correlated databases (Figure 7b), BNL, BTree and Index perform better that DC and Bitmap. Recall that correlated databases have fewer skyline points and this is advantageous to BNL, Index and BTree. Bitmap and DC, however, are much slower due to the overheads involved.

Figure 7c shows the results for independent databases. The relative performance of the various schemes is similar to the results on anti-correlated databases. The main difference is that BNL and DC have improved due to smaller number of skyline points. However, we note that both Bitmap and Index are still able to produce the first few tuples fairly quickly. Other experiments reported in [10] showed that Bitmap and Index perform even better for larger number of dimensions.

In summary, we believe that both Bitmap and Index are useful for progressive skyline computation. In particular, the performance of Index and its robustness to different types of databases of varying dimensions makes it an even more attractive option.

4.2.3 Experiment 3: Effect of buffer size

This experiment analyzes the effect of buffer space on the various algorithms using an anti-correlated 5-dimensional database. We varied the size of the main-memory buffers from 100 KB to 10 MB. We omit the results for BTree because prior results have already shown that it performs badly for this situation. Figure 8 shows the results when the buffer size is varied. From Figure 8, we can see that as the buffer increases

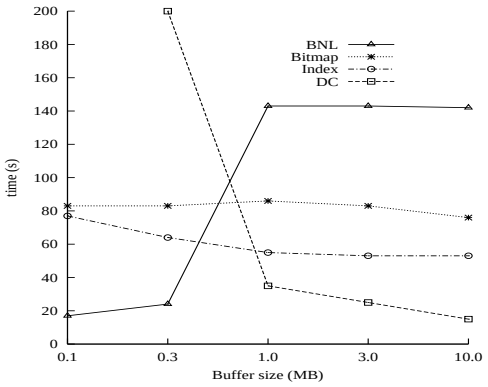
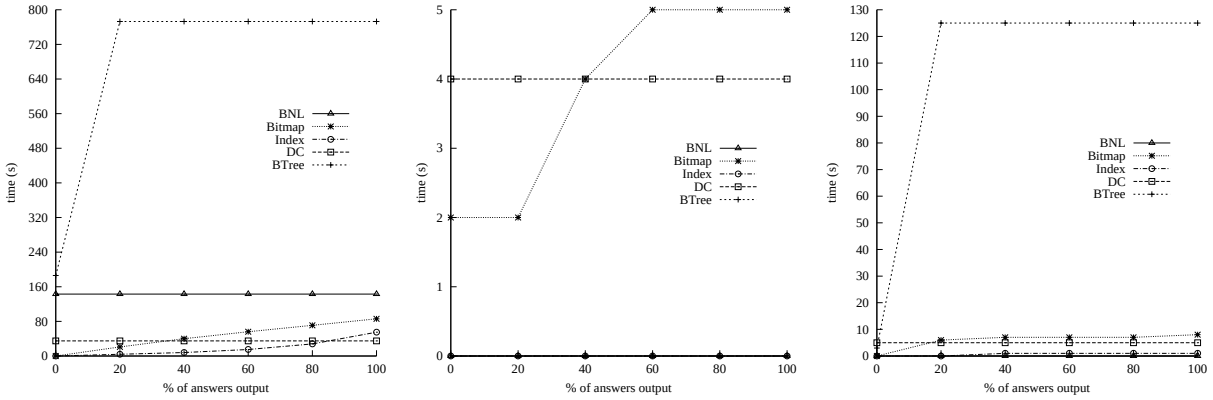


Figure 8: Varying the size of the buffer (anti-correlated database, 5 dimensions)

from 0.1 MB to 10 MB, DC's performance improves while BNL's performance degrades. Both results are consistent with [2]. On the other hand, the performance of Bitmap and Index remain fairly consistent. Although Bitmap can load more bitmaps into memory with a larger buffer, the overheads from loading and processing the bitmaps remain significant (due to the size of each bitmap). Hence, Bitmap only improves marginally. For Index, although the increase in memory results in more processing time (because more skyline points are now held in memory), we note that this effect is minimal, making Index a feasible option whether the memory is scarce or not.

4.2.4 Experiment 4: Effect of number of distinct values per dimension

In this experiment, we varied the number of distinct values each dimension of a tuple can take using an anti-correlated database of dimension 5 and 1 MB of main memory. We do not consider BTree since it is expected to perform badly. For completeness, we included the results where the number of distinct values is 100000 for BNL, Index and DC (we omit Bitmap for this case as it is expected to perform badly in this situation). Figure 9 shows the results of this experiment. From the results, we can see that as the number of distinct values increases, the performance of BNL and Bitmap become worse while the response times of DC and Index just increase slightly. First, when the number of distinct values is small, the number of skyline points decreases. This enables BNL to perform better than the rest. On the other hand, Index now has to process larger partitions, thereby incurring a runtime penalty. DC remains fairly consistent as it is independent of the number of distinct values in the datasets. Finally, Bitmap does not perform as well as we have expected although the runtime has reduced significantly. However, we can expect the performance of Bitmap to improve for even smaller sets of distinct values.



(a) Anti-Correlated (b) Correlated (c) Independent

Figure 7: Interval timings for dimension 5

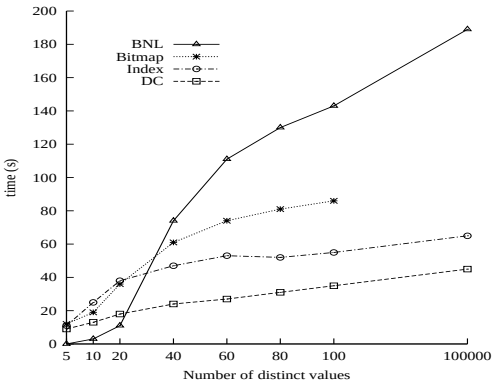


Figure 9: Varying the number of distinct values (anti-correlated database, 5 dimensions, 1 MB buffer)

5 Conclusion

In this paper, we have presented two novel algorithms to compute the skyline of a set of points. The main feature of the algorithms is that they can provide the skyline points progressively. The first algorithm, Bitmap, is completely non-blocking and exploits a bitmap structure to quickly identify whether a point is an interesting point or not. The second method, Index, exploits a transformation mechanism and a B^+-tree index to return skyline points in batches. Our extensive performance study showed that the proposed algorithms provide quick initial response time as compared to existing algorithms. Moreover, both schemes can also outperform the existing techniques in terms of total response time. While Index is superior in most cases, Bitmap performs well when the number of distinct values per dimension is small.

Acknowledgement

This work is partially supported by the University Research Grant RP982694. We would also like to thank the authors of [2] for providing us with the source of the data generator, which we adapted in our experimental study. Special thanks to the anonymous reviewers who provided very good insights that helped improve the technical quality and literary style of the paper.

References

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

[2] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001.

[3] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

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

[5] B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 166–174, 2000.

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

[7] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 1999.

[8] 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.

[9] 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.

[10] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. http://www.comp.nus.edu.sg/~engpk/pub/skyline.ps.

[11] C. Yu, B. C. Ooi, and K. L. Tan. Progressive knn search using b^+-tree. In *submitted for publication*, 2001.