Caching Dynamic Skyline Queries*

Dimitris Sacharidis[1], Panagiotis Bouros[1**], and Timos Sellis[1,2]

[1] National Technical University of Athens
9 Iroon Polytechniou, Athens 157 80, Greece
{dsachar,pbour}@dblab.ntua.gr
[2] Institute for the Management of Information Systems — R.C. Athena
17 G. Mpakou, Athens 115 24, Greece
timos@imis.athena-innovation.gr

Abstract. Given a query tuple q, the dynamic skyline query retrieves the tuples that are not dynamically dominated by any other in the data set with respect to q. A tuple dynamically dominates another, w.r.t. q, if it has closer to q's values in all attributes, and has strictly closer to q's value in at least one. The dynamic skyline query can be treated as a standard skyline query, subject to the transformation of all tuples' values. In this work, we make the observation that results to past dynamic skyline queries can help reduce the computation cost for future queries. To this end, we propose a caching mechanism for dynamic skyline queries and devise a cache-aware algorithm. Our extensive experimental evaluation demonstrates the efficiency of this mechanism compared to standard techniques without caching.

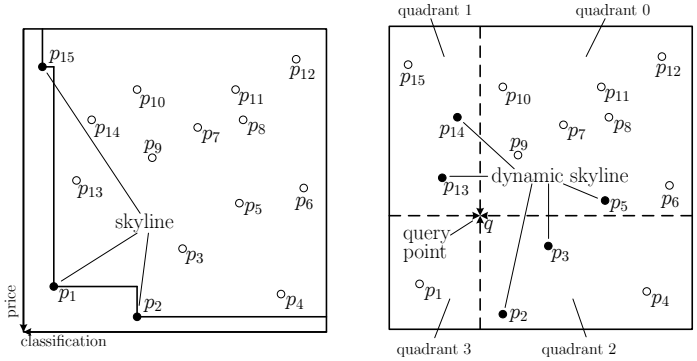
Keywords: skyline, dynamic skyline query, caching

1 Introduction

The skyline query has received considerable attention since its introduction in the database community [1]. Consider a data set P where each tuple is represented as a d-dimensional point. The *skyline query* returns all points in P not dominated by another point. A point p_i is said to *dominate* another point p_j if for all dimensions p_i has equal or smaller coordinate values than p_j and in at least one dimension p_i has strictly smaller value than p_j. Intuitively, assuming in all dimensions lower values are better, the skyline query retrieves the best tuples, irrespective of how a user assigns preference to each dimension. More formally, for any monotone preference function that assigns scores to tuples, the highest scored — most preferable — tuple is included in the skyline.

* This work has been funded by the project PENED 2003. The project is cofinanced 75% of public expenditure through EC - European Social Fund, 25% of public expenditure through Ministry of Development - General Secretariat of Research and Technology and through private sector, under measure 8.3 of OPERATIONAL PROGRAMME "COMPETITIVENESS" in the 3rd Community Support Programme.
** The author is partially supported by the Greek State Scholarships Foundation (IKY).

Consider a table that contains entries about hotels, with attributes Name, Price and Classification. Naturally, considering only its price, a hotel is more preferable if it is cheap. Similarly, regarding its classification, a high-starred hotel is more desirable. Figure 1(a) shows 15 hotels drawn as points in the two dimensional plane, where the x dimension is Classification, and the y is Price. The arrows in the axes indicate the direction of preference in each dimension, e.g., the more preferable high ranked (budget) hotels have lower x (y) values. Notice that there is no hotel that dominates p_1, p_2 and p_{15}, and, hence, they all belong in the skyline, as pointed in Figure 1(a). On the other hand, clearly, p_3 cannot be in the skyline as it is dominated by both p_1 and p_2. In fact, any hotel that resides in the right-hand side with respect to the line connecting the skyline hotels, is dominated.



(a) Skyline Query (b) Dynamic Skyline Query

Fig. 1. Skyline queries

A natural extension of the skyline query is its dynamic counterpart, introduced in [2]. Given a query point q, not necessarily in the data set P, the *dynamic skyline query* retrieves all points in P not dynamically dominated, with respect to q, by another point. A point dynamically dominates another, w.r.t. q, if it has closer to q's values in all dimensions, and has strictly closer to q's value in at least one. Returning to the hotel example, suppose that a user looks for hotels that match her budget and standards. For this reason, she specifies her "ideal" hotel q and wishes to retrieve all similar, in price and classification, hotels not dominated by others. Figure 1(b) illustrates the query point q and the dynamic skyline with respect to it. The dynamic skyline query w.r.t. q returns the hotels p_2, p_3, p_5, p_{13} and p_{14}, as shown in the figure. Notice that p_9 is not in the skyline, because p_{13} is closer to q in all dimensions, i.e., p_{13} matches the "ideal" hotel better than p_9 both in price and classification.

Dynamic skylines are useful in a variety of settings, where preferences are defined relatively to an exemplar, as in the ideal hotel scenario previously described. They also serve as the basic block for more complex queries. Seeing the skyline computation problem from the micro-economic perspective, other types

of dominance related queries [3] also make sense. For example, hotel owners might be interested to find out for which ideal hotels specified by users their hotels belong in the skyline. The latter is known as the reverse skyline problem [4]. Further, a query can specify a set of exemplars, rather than one, and the dominance relationships are adjusted accordingly to consider the entire set. Examples of such queries include the multi-source skyline [5] and the spatial skyline [6].

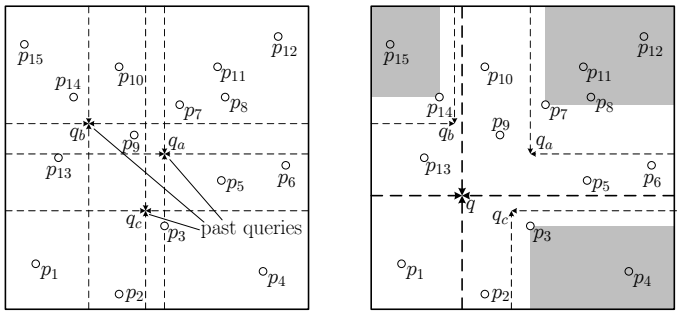
A dynamic skyline query can be reduced to a static skyline query, subject to the transformation of all points' coordinates. In particular, given query q, a point p is transformed to p' such that the i-th coordinate of p' is computed as $p'^i = |p^i - q^i|$. Therefore, any method designed for the standard skyline query can be trivially applied to the dynamic case. Note that all such methods are designed to solve a single instance, given a single data set. Therefore, in the dynamic case, where multiple instances — one for each query point — need to be solved, the algorithm must run anew each time, examining all points for dominance. In this paper we show that results to past dynamic skylines queries can help reduce the cost of processing future queries. We present a caching mechanism that maintains the most useful past results and uses them to exclude from consideration certain points.

Figure 2 illustrates the intuition behind our caching mechanism. Assume that q_a, q_b and q_c are past dynamic skyline queries, as depicted in Figure 2(a). Observe that each query point partitions the space into 4 quadrants. Let q represent the dynamic skyline query under consideration, shown in Figure 2(b), and examine the upper-right quadrant, which contains the past query q_a. Assuming we have cached the result for query q_a, we know that p_7 is part of q_a's dynamic skyline and that it dominates points p_8, p_{11} and p_{12}, as seen in the upper right shaded area in Figure 2(b). Furthermore, since p_7 lies in the same quadrant (upper-right) with respect to q_a as q_a lies with respect to q, we conclude that these points are dynamically dominated by p_7 w.r.t. q, as well. Indeed, p_8 is farther, in all dimensions, from both q_a and q than p_7. With analogous reasoning and by examining the past query q_b (q_c) one can deduce that p_{15} (p_4) cannot be in the dynamic skyline of q since it is dominated by p_{14} (p_3).

The contributions of this work can be summarized as follows:

1. We introduce the notion of *orthant skylines* and examine its relationship with dynamic skylines.
2. We extend the well-known Bitmap algorithm to compute the orthant skylines in parallel to the dynamic skyline, incurring small computation overhead.
3. We show how cached orthant skyline queries can help expedite the computation of future dynamic skyline queries. We propose three cache replacement policies for deciding which queries to expunge when the cache is full.
4. We perform an extensive experimental evaluation that demonstrates the efficacy of the caching mechanism, as portrayed by the significant reduction in query processing time in all settings.

Paper outline. First, in Section 2 we review literature on skyline related problems, focusing on the Bitmap algorithm, which is used throughout this work.



(a) Dynamic Skyline Queries (b) Pruning Points

Fig. 2. Caching skylines

Then, in Section 3 we present and formalize the basic notions discussed in this paper. The extension to the Bitmap algorithm for dynamic skylines and the caching mechanism are discussed in detail in Section 4. The most important experimental findings are presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Computing the points in the skyline, also known as finding the maxima in a set of vectors [7], has been thoroughly studied in the area of computational geometry where a large number of theoretical results exists. The first work to address the skyline computation problem in the context of databases was [1]. The authors discuss various techniques: they devise an algorithm that iterates over all points using block nested loops (BNL), propose a B-tree based approach, and also adapt the multidimensional divide and conquer algorithm [8] to handle external memory. An extension of the BNL algorithm that relies in presorting the points is introduced in [9]. The work in [10] introduces progressive algorithms that output points guaranteed to belong in the skyline without having to scan the entire data set. The Bitmap algorithm encodes all points using a bitmap representation and uses fast bitwise operations to extract the skyline points. We use Bitmap as the basis of our methodology for computing dynamic skylines in the presence of cache, and, thus, we present it in detail in Section 2.1. Another indexed method based on B-trees is also discussed in [10], where points are sorted according to their lowest valued coordinate.

Algorithms that use R-trees to index points have also been proposed. In [11] the authors observe that the nearest neighbor (NN) point to the beginning of the axes is always part of the skyline. This point segments the dataset into overlapping partitions according to its coordinates. Then, NN search is performed on each partition and the algorithm proceeds iteratively. Special care needs to be taken to remove duplicates resulting from the overlapping partitions. The branch

and bound algorithm (BBS) introduced in [2] avoids the pitfalls of the nearest neighbor approach. BBS maintains the expanded R-tree entries into a heap in ascending order of their minimum distance to the beginning of the axes. The first point visited in this manner is the NN and belongs to the skyline. When an entry is de-heaped, only its children not dominated by the skyline points found so far are inserted into the heap. BBS is proved to examine only the nodes in the R-tree that can potentially contain skyline points, and, hence, is I/O optimal.

The notion of dynamic skyline was first introduced in [2], where a variant of the BBS algorithm was presented. Given a point p, the reverse skyline query [4] retrieves the points whose dynamic skyline includes p. The authors in [4] present algorithms that are based on finding the global skyline of p, a notion related to the orthant skylines defined in this work. Another related notion is the multi-source skyline query [5, 6], in which a set of query points is specified and the result contains the points not dominated w.r.t. to the set.

When only a subset of the dimensions is considered, the skycube operator [12] returns the points that belong in the skyline. Some dominance related queries seen from the micro-economic perspective are presented in the data-warehouse framework of [3]. When the domain of a dimension is partially ordered, i.e., its values belong in a hierarchy, the skyline computation becomes more involved and the final result may require pruning as discussed in [13]. The notion of probabilistic skylines is defined [14] for the case where multiple tuples (or samples) correspond to randomly distributed objects in the data set.

2.1 The Bitmap Algorithm

The Bitmap algorithm was introduced in [10] for determining the skyline points efficiently when the domains of the defining dimensions are small and, most importantly, discrete. Briefly, Bitmap works as follows: (a) it pre-processes all points to obtain an appropriate bitmap representation, and (b) it checks each point for dominance against all points and outputs it if not dominated. The latter step is efficiently performed by fast bitwise AND/OR operations on the bitmap representations obtained in the former step.

Bitmap representation. For ease of presentation, we assume that all d dimensions have a domain of size n and its values belong in $\{0, 1, \ldots, n-1\}$, 0 being the most preferable value; the extension to dimensions with different domains is straightforward. A value u is represented by a bitmap of size n, where the u most significant bits are set to 0 and the remaining (the $n-u$ least significant bits) are set to 1. A d-dimensional point is, hence, represented as d bitmaps, one for each of its coordinates. We maintain the bitmap representation for all points in a bitmap table. For example, assume $n = 8$ and consider the point $p_1(0, 3, 7)$; its coordinates are represented as the bitmaps 11111111, 00011111 and 00000001, respectively. Figure 3(a) shows the bitmap table for 6 points, including that of p_1. The function of the bold and italicized bits will become apparent in the following.

	D_1	D_2	D_3
$p_1(0,3,7)$	11***1***1111	**0**0011111	000***0***0001
$p_2(6,7,2)$	00***0***0011	**0**0000001	001***1***1111
$p_3(1,0,2)$	01***1***1111	**1**1111111	001***1***1111
$p_4(3,0,4)$	00***0***1111	**1**1111111	000***0***1111
$p_5(2,2,6)$	00***1***1111	**0**0111111	000***0***0011
$p_6(6,4,7)$	00***0***0011	**0**0001111	000***0***0001

(a) Bitmap table

$A^1 = 101110 \quad B^1 = 101010$
$A^2 = 001100 \quad B^2 = 000000$
$A^3 = 011100 \quad B^3 = 011000$
$A \ = 001100 \quad B \ = 111010$

$C = A\&B = 001000$

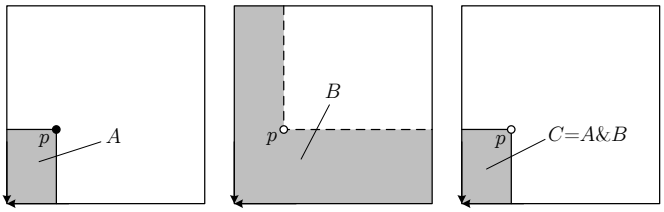
(b) Dominance check for p_4

Fig. 3. Bitmap example

Dominance check. The *dominance check* of point p identifies the points that dominate it. Clearly, a point belongs to the skyline if its dominance check identifies no dominating point. Performing the check in the Bitmap algorithm involves extracting vertical *bitslices* and performing bitwise AND/OR operations. Let $p^i \in \{0, 1, \ldots, n-1\}$ denote the i-th coordinate of p, where $1 \leq i \leq d$. We extract two vertical bitslices, A^i and B^i, from the bitmap representation of the data set. We extract a single bit for each point; thus, each bitslice has length N, where N denotes the size of the data set. In particular, we obtain the bitslice A^i by juxtaposing the (p^i+1)-th bit of the bitmap representation of the i-th coordinate for all points. Similarly, we obtain B^i by juxtaposing the preceding, i.e., the p^i-th, bit of the bitmap representation of the i-th coordinate for all points; note that when $p^i{=}0$ we explicitly set B^i to all zeros. Figure 3(b) shows the A^i, B^i bitslices for the dominance check of $p_4(3,0,4)$. For the A^i bitmaps we extract the 4th, 1st and 5th bit, shown in bold in Figure 3(a), for the first, second and third dimension of each point. For the B^i bitmaps we extract the 3rd and 4th bit, shown in italics in Figure 3(a), for the first and third dimension of each point; B^2 is set to all zeros. Given point p, its bitslice A^i encodes which points (i.e., those whose corresponding bit in A^i is set to 1) are equally as good or better than p with respect to the i-th dimension. In other words if the k-th bit of A^i is set to 1, then the k-th point has equally good or better value than p in the i-th coordinate. On the other hand, the bitslice B^i encodes the points that are strictly better in the i-th dimension.

Let $A = A^1 \& A^2 \& \ldots \& A^d$ denote the bitwise AND operation of all A^i bitslices. A indicates the points that are equally as good or better than p in *all* dimensions. Consider a point p in the 2 dimensional space shown in Figure 4. All points, including p, that reside in the shaded area of Figure 4(a) have their bit in A set to 1; for all other points the bit is 0. Similarly, let $B = B^1 | B^2 | \ldots | B^d$ denote the bitwise OR operation of all B^i bitslices. Then, B indicates the points that are strictly better than p in *at least one* dimension. All points that reside in the shaded area, excluding those in the dashed line and p, shown in Figure 4(b) have their bit in B set to 1; for all other points the bit is 0. According to the definition of dominance, if a point has its corresponding bit set both in A and B, then it dominates p, and, hence, p is not in the skyline. On the other hand, if $C = A\&B$ has no bit set, then p is not dominated by any point, and thus belongs in the skyline. All points, excluding p, that reside in the shaded area

shown in Figure 4(c) dominate p and thus have their bit in C set to 1; for all other points the bit is 0.



(a) Bitmap A (b) Bitmap B (c) Bitmap C

Fig. 4. Dominance check

Returning to the dominance check of point $p_4(3, 0, 4)$ for the example illustrated in Figure 3(b), notice that $A = 001100$ and $B = 111010$; hence, $C = 001000$. Since the third bit in C is set to 1, it follows that $p_3(1, 0, 2)$ dominates $p_4(3, 0, 4)$. So, p_4 does not belong in the skyline.

3 Preliminaries

In this section, we formally define the notions of dominance, and skyline points. We assume a d-dimensional space, where D_i denotes the domain of the i-th dimension, $i \in \{1, \ldots, d\}$. Each domain D_i is totally ordered by $<$ assigning preference to the values of the domain. Consider values $u, v \in D_i$; u is more preferable than v iff $u < v$. The data set P contains $N = |P|$ d-dimensional points. Each point $p \in P$ belongs in the space $D = D_1 \times D_2 \times \cdots \times D_d$ and is represented by its coordinates, $p = (p^1, p^2, \ldots, p^d)$.

Definition 1 (Dominance). *Let $p_1, p_2 \in P$ and $i, j \in \{1, \ldots, d\}$. A point p_1 dominates another point p_2, denoted as $p_1 \prec p_2$, iff (i) for all dimensions, p_1^i is more, or equally preferable than p_1^i, i.e., $\forall i : p_1^i \leq p_2^i$, and (ii) in at least one dimension, let j, p_1^j is strictly more preferable than p_2^j, i.e., $\exists j : p_1^j < p_2^j$.*

A point not dominated by any other in the data set is called a skyline point. Intuitively, one cannot prefer a non-skyline point over a skyline point for any preference function that is monotonic in each dimension. In other words, the skyline contains the top-1 point for any preference function and, conversely, for a given skyline point there always exists a function under which this point is the top-1.

Definition 2 (Skyline). *The skyline of P, denoted as $SL(P)$, is the set of points in P that are not dominated by any other point of P, i.e., $SL(P) = \{p_1 \in P \mid \nexists p_2 \in P : p_2 \prec p_1\}$.*

The skyline query retrieves the points that belong in the skyline. For example, for the data set shown in Figure 1(a), the skyline query retrieves the points p_1, p_2 and p_{15}.

3.1 Dynamic Skyline

According to the definitions of dominance and skyline presented above, the most preferable point is the beginning of the axes $o = (0, \ldots, 0)$, assuming it exists in P, since it dominates all other points. As argued in Section 1, however, in many cases the most preferable point could be a user specified point q. In this case we need to express the notions of preference and dominance relative to q. Given a point $q = (q^1, \ldots, q^d) \in D$ (not necessarily in P), the value $u \in D_i$, for some i, is more preferable than the value $v \in D_i$ iff $|u - q^i| < |v - q^i|$. Based on this preference notion, we now provide the definitions of dynamic dominance and skyline.

Definition 3 (Dynamic Dominance). *Let $p_1, p_2 \in P$, $i, j \in \{1, \ldots, d\}$ and $q \in D$. Given a query point q, a point p_1 dynamically dominates, w.r.t. q, another point p_2, denoted as $p_1 \prec_q p_2$, iff (i) for all dimensions, p_1^i is more, or equally preferable, w.r.t. q, than p_2^i, i.e., $\forall i : |p_1^i - q^i| \leq |p_2^i - q^i|$, and (ii) in at least one dimension, let j, p_1^j is strictly more preferable, w.r.t. q, than p_2^j, i.e., $\exists j : |p_1^j - q^j| < |p_2^j - q^j|$.*

Definition 4 (Dynamic Skyline). *Given a query point $q \in D$, the dynamic skyline of P w.r.t. q, denoted as $DSL(P, q)$, is the set of points in P that are not dynamically dominated, w.r.t. q, by any other point of P, i.e., $SL(P) = \{p_1 \in P \mid \nexists p_2 \in P : p_2 \prec_q p_1\}$.*

Consider the example data set and query q shown in Figure 5. The dynamic skyline w.r.t. q contains the points p_2, p_3, p_5, p_{13} and p_{14} drawn with black solid circles in the figure.

The dynamic counterparts of the dominance and skyline notions, essentially, correspond to the standard notions applied to the transformed data set P' obtained by mapping each point $p = (p^1, \ldots, p^d) \in P$ to the point $p' = (|p^1 - q^1|, \ldots, |p^d - q^d|)$, given the query point q. Consider Figure 5; observe that any point p, where $p^j - q^j < 0$ for at least one dimension, has been mapped to a point p' in the upper right quadrant w.r.t. q. The mapping is shown with a dashed line and the mapped point is drawn as a dashed circle.

As illustrated in Figure 5, the query point q partitions space D into the 4 quadrants (2^d orthants in the d-dimensional case) defined by constraining the space to be higher or lower than q^j for each dimension j. An orthant can be identified by a number written in binary containing d bits where the j-th bit is 0 (1) if for the j-th dimension the orthant contains the values not smaller (smaller) than q^j. Figure 5 shows the 4 orthants and their ids assuming dimension order yx. We introduce the notion of orthant skylines, which is defined as the dynamic skyline when considering only the points in P inside an orthant.

Definition 5 (Orthant Skyline). *Given a query point $q \in D$, the o-th orthant skyline of P w.r.t. q, where $o \in \{0, \ldots, 2^d - 1\}$, denoted as $OSL(P, q, o)$, is the set of points in P that belong to the o-th orthant and are not dynamically dominated, w.r.t. q, by any other point of that orthant.*

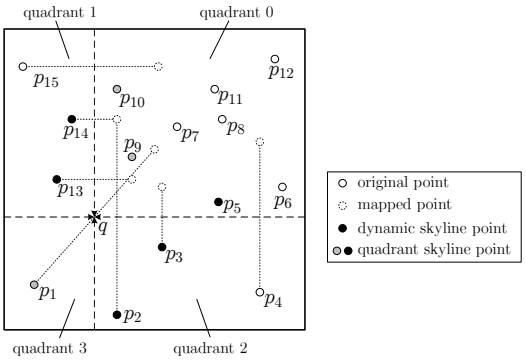


Fig. 5. Mapping points

Return to the example of Figure 5. The upper right quadrant (with id 0) skyline contains points p_5, p_9 and p_{10}, which are drawn with filled circles (black or grey). Note that an orthant skyline point can be dominated by the skyline point of another orthant and, hence, the former cannot belong in the dynamic skyline. For example, p_9 is dominated by the mapping of point p_{13} and, hence, is not part of the dynamic skyline. Similarly, p_{10} is dominated by the mappings of points p_2 and p_{14}, which happen to coincide. It is straightforward to see that the following lemma regarding the union of all orthant skylines w.r.t. q, termed global skyline in [4], holds.

Lemma 1. *The union of all orthant skylines w.r.t. q is a superset of the dynamic skyline w.r.t. q.*

4 Caching Dynamic Skylines

We first present the dynamic Bitmap algorithm, termed DBM, for obtaining the orthant skylines as well as the dynamic skyline in Section 4.1. Next, in Section 4.2, we demonstrate that caching queries and their orthant skylines can help reduce the execution time for future dynamic skyline queries. Finally we discuss cache replacement policies in Section 4.3.

4.1 Computing the Orthant Skylines

The DBM algorithm computes the orthant skylines and the dynamic skyline with respect to a query point q. Because the number of orthants is exponential to the dimensionality, it is crucial that our method finds the orthant skylines with little overhead compared to calculating only the dynamic skyline.

Initially, we construct the bitmap table for all points. In particular, we represent each coordinate of point $p \in P$ by converting the value $|p^i - q^i|$ (and not p^i) into the $|D_i|$ bits, as described in Section 2.1. We maintain a *global mask* M of length N to indicate the points that need to be considered. Initially all bits are set to 1. In addition, DBM creates the *orthant masks* of length N, denoted

M_j for $j \in \{0, \ldots, 2^d - 1\}$, to indicate which points belong to an orthant. Note that in the following we slightly abuse notation by referring to p's bit in M as $M[p]$. If p resides in the o-th orthant w.r.t. q, then $M_o[p]$ is set to 1. The orthant masks can be created in parallel to the bitmap table construction, as points are examined, by identifying the orthant a point resides in. Recall from Section 3.1 that the orthant id written in binary contains d bits, one for each dimension. Assume that point p is considered; the sign of $p^i - q^i$ designates the value of the i-th bit of p's orthant w.r.t. q, i.e., the bit is 0 if $p^i - q^i \geq 0$ and 1 if $p^i - q^i < 0$.

Figure 6 illustrates the DBM algorithm that examines each point in turn (Line 2). Given query point q, let p be the current point considered and let o denote the orthant p resides in w.r.t. q (Line 3). If p's bit in its orthant mask is set to 0, then we skip this point (Lines 4–5). Of course, in the case we examine here, this cannot happen, as the M_o mask is initialized to 1 for all its points; however, in Section 4.2, when the query cache is considered, this may no longer hold.

Next, DBM computes the C bitmap (Line 7) as discussed in Section 2.1 and performs two dominance checks, one for the orthant and one for the dynamic skyline. We distinguish three cases.

(I) $C\&M_o \neq 0$ denotes that p is dominated by some other point in its orthant (Line 8).
(II) $C \neq 0$ and $C\&M_o = 0$ denotes that p is not dominated by some other point in its orthant, but it is dominated by some point in another orthant (Line 11).
(III) $C = 0$ and $C\&M_o = 0$ denotes that p is not dominated by neither some other point in its orthant, nor by some point in another orthant.

In case (I), by Lemma 1, point p cannot belong to its orthant and the dynamic skyline. Hence, its bit in M and M_o is set to 0 (Lines 9–10). In case (II) p cannot be in the dynamic skyline but is part of it's orthant skyline. DBM sets its bit to 0 only in M (Line 12); its M_o bit remains set to 1. Finally, in case (III) DBM retains p's bit in M and M_o to 1. After all points have been examined, the M mask identifies the dynamic skyline points, whereas the $\{M_j\}$ masks identify the orthant skyline points (Lines 13–14).

4.2 Dynamic Skylines via Caching

In this section we show how caching of past queries and orthant skylines can help expedite dynamic skyline queries. We start by describing the cached Dynamic Bitmap algorithm, termed cDBM; we then discuss cache replacement strategies in Section 4.3. We assume that the cache Q stores for each past query q_i, the query itself and all its orthant skylines $OSL(P, q_i, j)$ for $j \in \{0, \ldots, 2^d - 1\}$. In particular, the orthant skyline $OSL(P, q_i, j)$ is represented as the N-length bitmap O_j^i in which p's bit is set to 1 if p is included in the j-th orthant skyline w.r.t q_i. Therefore, $Q = \{\langle q_i, \{O_j^i\}\rangle\}$, i.e., the cache needs to store 2^d bitmaps $\{O_j^i\}$ for each past query q_i; later, we provide a method to compress these bitmaps.

Algorithm DBM

```
   Input: data set P, query q, bitmap table T, masks M, {M_j}
   Output: orthant skyline bitmaps {O_j}, dynamic skyline bitmap R
 1 begin
 2     foreach p ∈ P do
 3         Let o be the orthant p resides in w.r.t. q
 4         if M_o[p] = 0 then                                   // pruned by cache
 5             continue
 6         else
 7             GetBitmapC(T, p)
 8             if C&M_o ≠ 0 then                                // case (I)
 9                 M_o[p] ← 0                                   // not in the orthant
10                 M[p] ← 0                                     // and not in the dynamic skyline
11             else if C ≠ 0 then                               // case (II)
12                 M[p] ← 0                                     // not in the dynamic skyline
13     R ← M
14     O_j ← M_j for all j
15     return ⟨R, {O_j}⟩
16 end
```

Fig. 6. The Dynamic Bitmap algorithm (DBM)

The intuition for using past orthant skylines lies in the fact that they can immediately and safely prune potentially large parts of the data set. Indeed, an orthant skyline contains precomputed information about the dominance checks in the particular orthant, as the next lemma suggests.

Lemma 2. *Consider queries $q_i, q \in D$ and a point $p \in P$, such that q_i belongs in the o-th orthant with respect to q, and p belongs in the o-th orthant w.r.t. q_i, and, thus, w.r.t. q as well. If p is not part of the o-th orthant skyline w.r.t. q_i, then, p is not part of the o-th orthant skyline w.r.t. q, and, hence, neither is part of the dynamic skyline w.r.t. q.*

Proof. Without loss of generality, assume $o = 0$. Since p is not part of the o-th orthant skyline w.r.t. q_i, it is dominated by (at least) one point, let p_a, i.e., $p_a \prec_{q_i} p$. Therefore, for $o = 0$ we have that $p_a^j - q_i^j \leq p^j - q_i^j$ for all j, and $p_a^k - q_i^k < p^k - q_i^k$ for at least one k, where $j, k \in \{0, \ldots, 2^d - 1\}$. Adding $q_i^j - q^j$ and $q_i^k - q^k$ to the previous inequalities, and since all quantities are non-negative, we obtain $p_a \prec_q p$. □

Figure 2(b) demonstrates Lemma 2 for the current query q and the past query q_a, which lies in the upper-right quadrant with respect to q. The skyline of the upper-right quadrant w.r.t. q_a contains a single point, p_7, which dynamically dominates p_8, p_{11} and p_{12} w.r.t. q_a. It is obvious that p_7 also dominates p_8, p_{11} and p_{12}, w.r.t. q, and hence, these, points cannot belong to the upper-right quadrant or dynamic skyline of q.

The cDBM algorithm for calculating the dynamic and orthant skylines is illustrated in Figure 8. For each query, cDBM first computes the masks (Line 4) and then calls the DBM algorithm (Line 5). Finally, the orthant skylines are inserted into the cache (Line 6).

The most important step of the cDBM algorithm is the ComputeMasks procedure, shown in Figure 7. Given query q, this procedure creates the masks M

and $\{M_j\}$ applying Lemma 2 to determine which points need not be considered. Initially, the cache Q is partitioned into sets Q_j for $j \in \{0, \ldots, 2^d - 1\}$, such that Q_j contains the queries that reside in the j-th orthant w.r.t. q (Line 2). Then, each point p is examined in turn (Line 3). The bitmap representation of p with respect to q is computed and the bitmap table T is updated (Line 8), as discussed in Section 4.1. Let o denote the orthant p lies w.r.t. q (Line 4). Then, p's bit in the o-th orthant mask is set to 1, whereas in all other orthant masks it is set to 0 (Lines 6–7); of course, p's bit in M is set to 1 (Line 5). The algorithm continues by examining the past queries that reside in the o-th orthant, i.e., those in Q_o. Let q_i be such a query (Line 9) and let o_i be the orthant that p lies in with respect to q_i (Line 10). If p lies in the same orthant with respect to q_i as q_i lies w.r.t. q, i.e., $o_i = o$ (Line 11), then Lemma 2 applies (Lines 12–16). Therefore, if p was not in o-th orthant skyline w.r.t q_i (Line 13), then it can be excluded from consideration in the orthant (Line 14) and the dynamic skyline (Line 15) w.r.t. q. If this was the case, then no other past query needs to be examined (Line 16).

Compressing Cached Queries. The caching mechanism discussed above has a large space overhead, as it requires storing 2^d bitmaps for each query in the cache. We address this issue making the following observation: a point can belong to only one orthant and, thus, can be part of only one orthant skyline per query. Given query q_i and its orthant skyline bitmaps O^i_j, for all $j \in \{0, \ldots, 2^d - 1\}$, we construct a single orthant skyline bitmap O^i by disjuncting all O^i_js. Then, p's bit in O^i is set to 1, if point p belongs in the skyline of its orthant w.r.t. q. Note that the ComputeMasks procedure need not change; in Line 13 of Figure 7, the O^i mask can be used instead of the o-th orthant mask O^i_o.

Procedure ComputeMasks

```
   Input: data set P, query cache Q, query q
   Output: bitmap table T, masks M, {M_j}
 1 begin
 2     partition Q to the {Q_j} sets w.r.t. q
 3     foreach p ∈ P do
 4         Let o be the orthant p resides in w.r.t. q
 5         M[p] ← 1                                   // initialize mask to 1
 6         M_o[p] ← 1                                 // initialize o-th orthant mask to 1
 7         M_j[p] ← 0, for all j ≠ o                  // initialize all other orthant masks to 0
 8         update T with BitmapEncode(p, q)
 9         foreach ⟨q_i, {O^i_j}⟩ ∈ Q_o do
10             Let o_i be the orthant p resides in w.r.t. q_i
11             if o_i = o then
12                 Let O^i_o be the o-th orthant skyline of q_i from {O^i_j}
13                 if O^i_o[p] = 0 then               // if p not in its orthant skyline w.r.t. q_i
14                     M[p] ← 0                       // it cannot be in the dynamic skyline w.r.t. q
15                     M_o[p] ← 0                     // neither in the j-th orthant skyline w.r.t. q
16                     break
17     return ⟨T, M, {M_j}⟩
18 end
```

Fig. 7. Computing masks given the query cache

Algorithm cDBM

```
   Input: data set P
1  begin
2  |  Q = ∅
3  |  foreach incoming q do
   |  |  // initialize masks and construct bitmap table
4  |  |  ⟨T, M, {M_j}⟩ ← ComputeMasks(P, Q, q)
   |  |  // calculate dynamic and orthant skylines
5  |  |  ⟨R, {O_j}⟩ ← DBM(P, q, T, M, {M_j})
6  |  |  update Q with ⟨q, {O_j}⟩          // run a cache replacement policy
   |  |
7  end
```

Fig. 8. The cached Dynamic Bitmap algorithm (cDBM)

4.3 Cache Replacement Policies

In this section we discuss replacement policies for our caching mechanism. The objective of these policies is the identification of the least *useful* query point among those in the cache that must be discarded together with its orthant mask. The first two policies we consider are the common Least Recently Used (LRU) and Least Frequently Used (LFU) policies, which keep track of the usage for each query in the cache. On the other hand, the Least Pruning Power (LPP) policy measures the pruning ability of each query and discards the least strong.

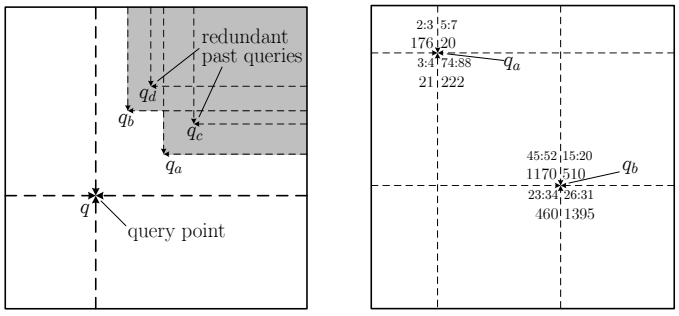
LRU and LFU policies. In Section 4.2 we used *all* queries in the cache to discover which points to exclude from consideration. However, given a query q, some of the queries in the cache are redundant, i.e., they identify points that can be pruned even if we don't consider these cached queries. This is exemplified in Figure 9(a), where q denotes the current query under consideration, and q_a, q_b, q_c and q_d are past queries in the cache that reside in the upper-right quadrant w.r.t. q. For each of these queries, their upper-right quadrant skyline can be used to prune some of the points that are contained in the dashed box ranging from the query to the upper right point in Figure 9(a). Observe that since q_c's box is entirely contained in q_a, the points the former can possibly prune can also be pruned by the latter; the same holds for q_d and q_b. It is easy to show that, given a query q, if a cached query q_c is dominated by another cached query q_a, then q_c can be safely disregarded when computing the masks for query q.

The previous observation suggests the following change to the ComputeMasks procedure in Figure 7. In Line 2, ComputeMasks partitions the cached queries according to the orthant they belong with respect to the query point q. Instead, we calculate the orthant skylines w.r.t. q of all queries in Q so that the set Q_j now contains the queries in Q that belong in the j-orthant skyline (and not all queries in the j-th orthant).

Consider the case when query point q is considered and that we compute the Q_j sets. In the Least Recently Used (LRU) policy, each time a cached query belongs in the orthant skyline, i.e., it will be used to prune points, we annotate it with a timestamp to indicate that it was used for query q. Note that the timestamp can be a simple counter that increases with each query considered. When the cache has reached its capacity we choose to evict the cached query that was least recently used, i.e., has the smallest timestamp. To efficiently

identify the cached query to be evicted, we maintain a priority queue with key the timestamp of each query. Therefore, updating the timestamp of an already stored query, inserting a new and evicting the least recently used one require time logarithmic to the size of the cache.

In the Least Frequently Used (LFU) policy, we maintain a usage counter for each query in the cache. Each time a cached query belongs in the orthant skyline w.r.t. the query point, i.e., it will be used to prune points, we increment its usage counter. We choose to always include in the cache the current query under consideration, and if the cache is full we evict the least frequently used query, i.e., that with the smallest usage counter. As before, a priority queue can be used to expedite the identification of the query to be evicted.



(a) Redundant cached queries (b) Pruning power

Fig. 9. Cache Replacement Policies

LPP policy. Intuitively, a useful cached query is one which has great pruning power, i.e., it can discard a large number of points for a lot of queries. Depending on the position of a cached query q_a relative to a query q, the pruning power of q_a can vary significantly. Let dp_a^j denote the number of points dominated by cached query q_a's j-th orthant skyline. Consider for example the two cached queries illustrated in Figure 9(b). If a query lies in the upper-left quadrant with respect to q_a, then q_a's skyline for the lower-right quadrant can prune 74 points, as indicated by the first of the three numbers per query and quadrant shown in Figure 9(b), i.e., $dp_a^3 = 74$. On the other hand, if a query lies in the lower-right quadrant w.r.t. q_a, then q_a's upper-left quadrant skyline can only prune 2 points, i.e., $dp_a^1 = 2$.

The pruning power of a query's orthant also depends on the probability of a query residing in the antisymmetric orthant. In the example of Figure 9(b), it is rather unlikely for a query to belong in the upper-left quadrant w.r.t. q_a, hence q_a's highly dominant quadrant will rarely be used. It is reasonable to assume that queries follow a similar distribution to the data set; therefore the probability of a query residing in any area is analogous to the number of data points this area includes. Given a cached query q_a, let np_a^j denote the number of points residing

in the j-th orthant w.r.t. q_a. In Figure 9(b), np_a^j is shown as the second number in the triad of numbers for all queries and quadrants.

Given a query q_a, its pruning power for the j-th orthant, denoted as pp_a^j, is given by $pp_a^j = np_a^{\bar{j}} \cdot dp_a^j$, where $\bar{j}$ identifies j's antisymmetric orthant. In Figure 9(b), pp_a^j is shown as the third number typed in larger font for each query and quadrant. In the case of query q_a, for example, the upper-left quadrant skyline dominates 2 out of 3 points, and the lower-right quadrant contains a total of 88 points; hence, the pruning power of the q_a upper-left quadrant is $2 \cdot 88 = 176$. The pruning power pp_a of a query q_a is the sum of its pruning power for all orthants, i.e., $pp_a = \sum_j pp_a^j$. Intuitively, a low pruning power implies that the query is expected to prune only a few points. LPP always evicts from cache the query with the least pruning power. As before, a priority queue can be used.

Note that a query's pruning power for all orthants can be computed with little overhead. The number of points in an orthant np_a^j can be counted in the ComputeMasks procedure while examining where each point resides with respect to the query. Also, the number of points dominated by an orthant skyline dp_a^j can be calculated as the number of points in the orthant minus the orthant's skyline size; the latter can be found by simply using a counter in the DBM algorithm.

5 Experimental Evaluation

We present an extensive experimental evaluation of the cDBM algorithm paired with the three cache replacement policies discussed in Section 4.3. In particular, we compare LRU, LFU, LPP with the Bitmap algorithm adapted to dynamic skyline query processing, denoted as NO-CACHE since it corresponds to the case where no cache is used. All algorithms are implemented in C++, compiled with gcc and executed on a 3 Ghz Intel Core 2 Duo CPU.

We use the generator from [15] to create data sets of three types of distribution:

- Independent: The attribute values are drawn from a uniformly random independent distribution.
- Correlated: Tuples whose attribute values are low, i.e., preferable, in one dimension have most likely low values in the other dimensions, as well.
- Anti-Correlated: Tuples whose values are low in one dimension have most likely high, i.e., not preferable, values in the other dimensions.

The dimensionality of the data set varies from $d = 2$ up to 6, where each dimension's domain contains a fixed number of discrete values, ranging from $|D| = 10$ up to 50. The size of the data set, N, is between 10 thousand and up to 100 thousand tuples. We test all cache replacement policies for different cache sizes that extend from $|Q| = 10$ to 50 queries. We perform $|Q| + 20$ dynamic skylines queries and we measure the average performance of all policies for the last 20 queries, so that the query cache is full in all cases. More specifically, we measure the average running time for each method and we count the average number

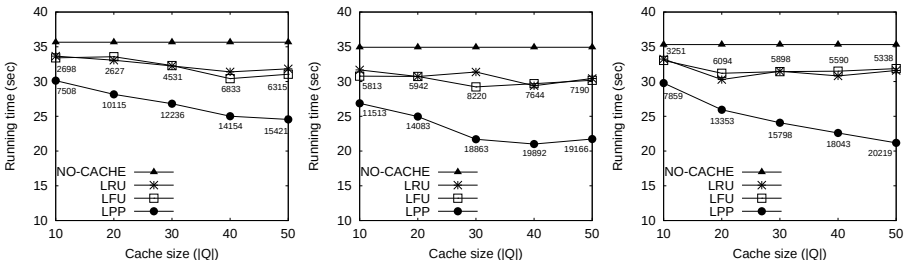
of points pruned by the cache. In each experiment we vary a single parameter while we set the remaining ones to their default values. Table 1 summarizes the parameters involved and their ranges; the default values are shown in bold.

Parameter	**Values**
N	10000, 20000, **50000**, 100000
d	2, 3, **4**, 5, 6
$\|D\|$	10, **20**, 50
$\|Q\|$	10, 20, **30**, 40, 50

Table 1. Experimental parameters

5.1 Experimental Results

In the first set of experiments we vary the cache size from $|Q| = 10$ to 50 while the data set is fixed to containing $N = 50000$ tuples with $d = 4$ attributes of cardinality $|D| = 20$. In this setting, the dataset is 1000 KB, whereas the cache size increases from 62.5 KB (6% of data size) for $|Q| = 10$, to 187.5 KB (19% of data size) for the default setting ($|Q| = 30$), and up to 312.5 KB (31% of data size) for the largest setting $|Q| = 50$. We measure the average running time and the average number of points pruned for 20 dynamic skyline queries when the cache is full.



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

Fig. 10. Varying the query cache size

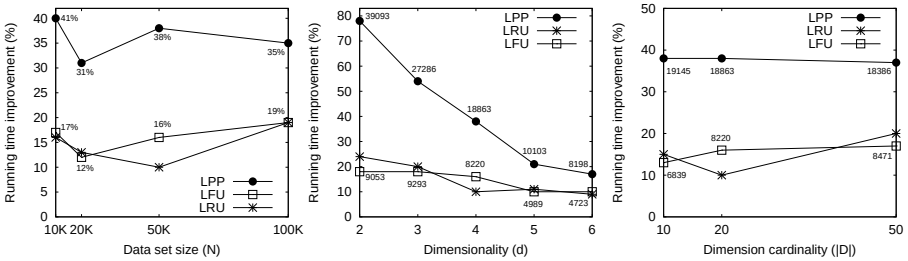
Figure 10 presents the results of all cache replacement policies for the three data sets. The average number of pruned points are shown next to the time measurements for the LPP and LFU policies. We also draw the running time when the queries are processed without cache, denoted as NO-CACHE, which is a straight line over $|Q|$. The expected behavior when the cache size increases is the number of pruned points to also increase, resulting in shorter running times. This is clearly the case for the LPP policy which outperforms the usage-based policies.

LRU and LFU, especially in the Anti-Correlated data set (Figure 10(c)), fail to take advantage of the larger cache. The reason for this behavior can be attributed to the fact that caching more queries recently (LRU) or frequently used (LFU) does not guarantee that future queries will benefit from them. In other words, the queries already seen are not representative of the queries to follow. On the other hand, LPP keeps in cache queries with great pruning power that can prove useful for *any* future query. For the maximum setting $|Q| = 50$, LPP immediately prunes 15421, 19166 and 20219 out of the 50000 points and decreases the processing time by 31%, 38% and 40% for the Independent, Correlated and Anti-Correlated data sets, respectively.

Figure 11 shows the effect of the distribution parameters on the caching mechanism. In this setting we draw the relative improvement in running time for the three cache replacement policies over the case of no cache, for the Correlated data set; similar results hold for the other distribution types. In Figure 11(a) we vary the data set size while keeping all parameters, including cache size Q, to their default values shown in Table 1. This implies that relative to the data size, the cache decreases as N grows. Still, Figure 11(a) shows that the policies can prune a rather significant part of the dataset (31% – 41% for LPP as shown by the labels in the figure), which is translated to an analogous running time improvement. Note that as the data set becomes denser, LFU and LRU's performance also increases.

In Figure 11(b) we vary the dimensionality of the data set, while the remaining parameters have their default values. The LPP policy is highly affected by the curse of dimensionality, i.e., as the space becomes sparser (since N is fixed) its pruning power rapidly decreases, i.e., from 78% down to 17%. The usage-based policies are also affected but to a lesser degree.

Finally, in Figure 11(c), we vary the domain cardinality for each dimension, when $N = 50000$, $d = 4$ and $|Q| = 50$. Larger $|D|$ values result in sparser data sets. However, unlike Figure 11(b), the cache replacement policies are not significantly affected. LPP in all cases improves running time by 38%.



(a) Varying N (b) Varying d (c) Varying $|D|$

Fig. 11. Effect of distribution parameters

6 Conclusions and Future Work

In this paper we study the problem of dynamic skyline queries from a fresh perspective. We consider the case where we keep past queries and their results in a cache so as to expedite future query processing. For this end, we introduce the notion of orthant skylines and extend a well-known skyline algorithm to handle them. Then, we prove that results of orthant skyline queries can potentially exclude a large part of the data set from the costly dominance checks. We propose three cache replacement policies so that the cache always contains the most useful queries and their results. Through extensive experimental results on synthetically generated data set, we demonstrate the efficiency of the proposed caching mechanism: using less than 20% of the data size for the cache, we can reduce the processing time by 40%. In the future, we plan to apply the caching framework to other skyline processing algorithms, focusing on indexed approaches. Furthermore, we will investigate methods to increase the pruning power of the cached queries while at the same time reducing the space overhead.

References

1. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE. (2001) 421–430
2. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM Transactions on Database Systems **30**(1) (2005) 41–82
3. Li, C., Ooi, B.C., Tung, A.K.H., Wang, S.: Dada: a data cube for dominant relationship analysis. In: SIGMOD Conference. (2006) 659–670
4. Dellis, E., Seeger, B.: Efficient computation of reverse skyline queries. In: VLDB. (2007) 291–302
5. Deng, K., Zhou, X., Shen, H.T.: Multi-source skyline query processing in road networks. In: ICDE. (2007) 796–805
6. Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: VLDB. (2006) 751–762
7. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. Journal of the ACM **22**(4) (1975) 469–476
8. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer (1985)
9. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE. (2003) 717–816
10. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: VLDB. (2001) 301–310
11. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: VLDB. (2002) 275–286
12. Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J.X., Zhang, Q.: Efficient computation of the skyline cube. In: VLDB. (2005) 241–252
13. Chan, C.Y., Eng, P.K., Tan, K.L.: Stratified computation of skylines with partially-ordered domains. In: SIGMOD Conference. (2005) 203–214
14. Pei, J., Jiang, B., Lin, X., Yuan, Y.: Probabilistic skylines on uncertain data. In: VLDB. (2007) 15–26
15. Random dataset generator for SKYLINE operator evaluation: http://randdataset.projects.postgresql.org/