Efficient Updates for Continuous Skyline Computations*

Yu-Ling Hsueh† Roger Zimmermann‡ Wei-Shinn Ku§

†Dept. of Computer Science, University of Southern California, Los Angeles, CA 90089
‡Computer Science Department, National University of Singapore, Singapore 117543
§Dept. of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849
{hsueh@usc.edu, rogerz@comp.nus.edu.sg, weishinn@auburn.edu}

Abstract. We address the problem of maintaining *continuous skyline queries* efficiently over dynamic objects with d dimensions. Skyline queries are an important new search capability for multi-dimensional databases. In contrast to most of the prior work, we focus on the unresolved issue of frequent data object updates. In this paper we propose the *ESC* algorithm, an **E**fficient update approach for **S**kyline **C**omputations, which creates a pre-computed *second skyline* set that facilitates an efficient and incremental skyline update strategy and results in a quicker response time. With the knowledge of the *second skyline* set, *ESC* enables (1) to efficiently find the substitute skyline points from the *second skyline* set only when removing or updating a skyline point (which we call a first skyline point) and (2) to delegate the most time-consuming skyline update computation to another independent procedure, which is executed after the complete updated query result is reported. We leverage the basic idea of the traditional *BBS* skyline algorithm for our novel design of a two-threaded approach. The first skyline can be replenished quickly from a small set of second skylines - hence enabling a fast query response time - while de-coupling the computationally complex maintenance of the second skyline. Furthermore, we propose the *Approximate Exclusive Data Region* algorithm (*AEDR*) to reduce the computational complexity of determining a candidate set for second skyline updates. In this paper, we evaluate the *ESC* algorithm through rigorous simulations and compare it with existing techniques. We present experimental results to demonstrate the performance and utility of our novel approach.

1 Introduction

Skyline query computations are important for multi-criteria decision making applications and they have been studied intensively in the context of spatio-temporal databases. Skyline queries have been defined as retrieving a set of points, which are not dominated by any other points. An object p dominates p', if p has more favorable values than p' in all dimensions. Some of the prior work on

* This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0534761, NUS AcRF grant WBS R-252-050-280-101/133 and equipment gifts from the Intel Corporation, Hewlett-Packard, Sun Microsystems and Raptor Networks Technology. We also acknowledge the support of the NUS Interactive and Digital Media Institute (IDMI).

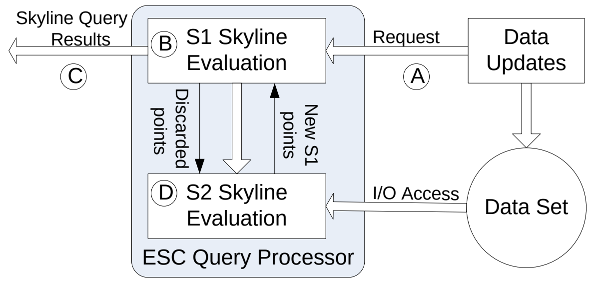


Fig. 1. ESC system framework

skyline queries assumed that data objects are static [13, 15]. Other approaches assumed that the skyline computation involved only a partial of dynamic dimensions [4]. In this paper, we address *Efficient Updates for Continuous Skyline Computations* over dynamic objects (*ESC* for short), where objects with d dynamic dimensions move in an unrestricted manner. Each dimension represents a spatial or non-spatial value. Towards an efficient continuous skyline computation the following challenges must be addressed: an effective incremental skyline query result update mechanism that is needed provides a fast response time of reporting the current query results, and an efficient strategy to reduce the search space dimensionality is required.

Existing work [6, 14, 19] generally computes a number of data point subsets, each of which is exclusively dominated by one skyline point. Therefore, when a skyline point moves or is deleted, only its exclusively dominated subset must be scanned. The determination of such an exclusive data set is very computationally complex in higher dimensions and it incurs a serious burden for the system in a highly dynamic environment. Therefore, these systems are often unable to provide up-to-date query results with a quick response time. We propose the *ESC* algorithm to efficiently manage the query results by delegating the time-consuming skyline update computations to another independent procedure, which is processed after the query processor reports the latest skyline query results. The key idea is to maintain a *second skyline* (or $S2$) set which is a skyline candidate set pre-computed when a traditional skyline (which we refer as the *first skyline*, $S1$) point requests an update. With the knowledge of the second skyline set, the skyline query result can be updated within a limited search space and the expensive computations (e.g., searching for new second skylines to substitute a promoted second skyline point) can be decoupled from the first skyline update computations.

Figure 1 shows the framework of the *ESC* system. The query processor initially computes the first and second skyline points. Any updates **(A)** performed on the data set are also submitted to the query processor. First, Task **(B)** examines whether the update request (e.g., inserting or removing a data point) affects the first skyline set. If the request point becomes a new $S1$ point, Task **B** inserts the new $S1$ point into the current $S1$ set and removes the current skyline points that are dominated by the new $S1$ point. These discarded $S1$ points (new

$S2$ points) are processed by Task **(D)** later to update the $S2$ set. In case that an update request stems from a removed or moving $S1$ point, some exclusive points are left un-dominated. The query processor searches for new substitute $S1$ points only from the $S2$ set. The query results **(C)** are immediately output as soon as Task **(B)** is completed. The processing time of the sequence of Tasks **(A)(B)(C)** is the system response time to a skyline query update. Task **(D)** maintains the $S2$ points when any $S2$ point is inserted or removed. To enhance Task **(D)**, which involves the expensive computation of determining exclusive data points where **(D)** searches for new or substitute $S2$ points from the rest of the data set, we also propose an *approximate exclusive data region* computation with lower amortized cost than existing techniques [14, 19]. The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 presents and details our continuous skyline query processing design. We extensively verify the performance of our technique in Section 4 and finally conclude with Section 5.

2 Related Work

Borzsonyi et al. [1] proposed the straightforward non-progressive *Block-Nested-Loop* (BNL) and *Divide-and-Conquer* (DC) algorithms. The BNL approach recursively compares each data point with the current set of candidate skyline points, which might be dominated later. BNL does not require data indexing and sorting. The DC approach divides the search space and evaluates the skyline points from its sub-regions, respectively, followed by merge operations to evaluate the final skyline points. Both algorithms may incur many iterations and they are inadequate for on-line processing. In [17], Tan et al. presented two progressive processing algorithms: the *bitmap* approach and the *index* method. *Bitmap* encodes dimensional values of data points into bit strings to speed up the dominance comparisons. The index method classifies a set of d-dimensional points into d lists, which are sorted in increasing order of the minimum coordinate. Index scans the lists synchronously from the first entry to the last one. With the pruning strategies, the search space is reduced. The *nearest neighbor* (NN) method [5] indexes the data set with an R-tree. NN utilizes nearest neighbor queries to find the skyline results. The approach repeats the query-and-divide procedure and inserts the new partitions that are not dominated by some skyline point into the to-do list. The algorithm terminates when the to-do-list is empty. In [13], a *branch and bound skyline* (BBS) algorithm traverses an R-tree to find the skyline points. Although BBS outperforms the NN approach, the performance can deteriorate due to many unnecessary dominance checks. Finally, many of the recent techniques aim at continuous skyline support for moving objects and data streams. Lin et al. [8] present n-of-N skyline queries against the most recent n of N elements to support on-line computation against sliding windows over a rapid data stream. Morse et al. [11] propose a scalable *LookOut* algorithm for updating the continuous time-interval skyline efficiently. Sharifzadeh et al. [16] introduce the concept of Spatial Skyline Queries (SSQ).

Given a set of data points P and a set of query points Q, SSQ retrieves those points of P which are not dominated by any other point in P considering their derived spatial attributes to query points in Q. For moving query points, a continuous skyline query processing strategy is presented in [4] with a kinetic-based data structure. However, prompt query response is not considered in the design. A suite of novel skyline algorithms based on a Z-order curve [3] is proposed in [6]. Among the solutions, *ZUpdate* facilitates incremental skyline result maintenance by utilizing the properties of Z-order curve. Other related techniques can be found in the literature [2, 19, 9, 12, 18]. However, all the aforementioned studies differ from the main goal of this research – supporting frequent skyline data object updates efficiently while providing a quick response.

3 ESC Algorithm

3.1 The Problem Definition of Continuous Skyline Queries

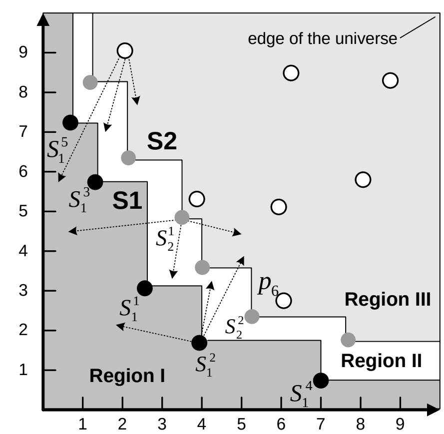


Fig. 2. $S1$ and $S2$ sets

The formal definition of skyline points in d-dimensional space is a distinct object set P, where any two objects $p = (x_1, ..., x_d)$ and $q = (y_1, ..., y_d)$ in the set satisfy the condition that if for any $k, x_k < y_k$, there exists at least one dimension of $m \leq d$ that satisfies $x_m > y_m$. We say p dominates q ($p \vdash q$ for short), *iff* $x_k < y_k$, $\forall k$ $(1 \leq k \leq d)$. The general setup of the problem consists of a set of dynamic query and data objects with d dimensions. Moving objects can freely move in an unrestricted and unpredictable fashion, meaning that their parameters x_k may arbitrarily change their values. The major challenging issue of a continuous skyline query is to avoid unnecessary dominance checking on irrelevant data points for skyline query result updates. After observing the *BBS* algorithm [13], we deduced that when evaluating the skyline query result, a set of *second skyline* ($S2$) points can always be obtained with little extra work while retrieving the *first skyline* ($S1$) points. We refer to the traditional skyline query result as the *first skyline*, consisting of $S1 = \{s_1^1, ..., s_1^m\}$. The *second skyline* $S2 = \{s_2^1, ..., s_2^k\}$ is defined as follows:

Definition 1: A data point p is a second skyline point *iff* $p \in (P - S1)$ and $\nexists p' \in (P - S1 - p)$, $p' \vdash p$. Informally, all $S2$ points are dominated by $S1$ and the rest of the data points $(P - S1 - S2)$ are dominated by both $S1$ and $S2$.

When a $S1$ point s_1^i is removed or at least one value of its dimensions changes, the $S2$ points are naturally considered as new $S1$ point candidates to "substitute" s_1^i. The features of a $S2$ set are as follows: (1) it is a pre-computed set that covers all the new $S1$ candidate points, and (2) $S2$ is a relatively small data set. Therefore, with the knowledge of $S2$, the query processor can efficiently update the query result and provide a quicker response time to the query point. An example is shown in Figure 2. If a $S1$ point s_1^2 moves to Region I, the search space for *ESC* to update the query result only involves the $S1$ set and the $S2$ set. In this case, s_1^2 remains a $S1$ point, but it dominates s_1^1. *ESC* needs to remove s_1^1 from the $S1$ set and s_1^1 becomes a new $S2$ point, since no existing $S2$ point can dominate it. Due to the movement of s_1^2, *ESC* searches for new $S1$ points from the $S2$ set. Since s_2^2 (an exclusive data point) is left un-dominated, s_2^2 becomes a new $S1$ point and is removed from the $S2$ set. The *ESC* algorithm delegates the necessary $S2$ maintenance (an independent procedure from $S1$ updates) to the query processor after $S1$ updates are completed. For example, new $S2$ points must be retrieved to substitute s_2^2. To avoid scanning through the entire data points in Region III for new $S2$ points, we propose an *approximate exclusive data region* (*AEDR*) computation in contrast to a traditional *exclusive data region* (*EDR*) computation. Based on our observation and analysis, we provide the lemmas for incrementally updating the skyline query results in the following sections. Table 1 summarizes the symbols and functions we use throughout the following sections.

Symbols	**Descriptions**
P	Number of data objects
d	Number of dimension
$S1$	First skyline point set (traditional skyline query result set)
$S2$	Second skyline point set
DataRtree	Disk-based Rtree for indexing P
S1Rtree	Main-memory Rtree for indexing $S1$ points
S2Rtree	Main-memory Rtree for indexing $S2$ points
$EDR(p)$	A set of data points in the exclusive data region
$AEDR(p)$	A set of data points in the approximate exclusive data region
$W(p)$	A set of skyline points in the dominance area of p
$p.DomArea$	The dominance area of p

Table 1. Symbols and functions

3.2 Second Skyline Computation

The existing work [14, 19] performs the time-consuming exclusive data point computations for the skyline query result updates. In Figure 3, the gray areas represent the traditional *EDRs* that contain exclusive data points. An *EDR* is not usually pre-computed because of the complexity of the calculation. In contrast, since the $S2$ points (new $S1$ candidates) can be easily computed before any $S1$ point issues an update, the query processor is able to satisfy a query request with the latest query result and with a quicker response time. To further reduce the search space of visiting $S2$ points to update the skyline query result, we introduce and define a *dominance set* for each $S1$ point s_1^i. A *dominance set* contains a group of $S2$ points which are dominated by s_1^i (denoted by $D(s_1^i)$) to substitute a removed or moving s_1^i point when dominance relationship has changed. For example in Figure 3 the dominance set of s_1^2 includes s_2^2. If s_1^2 is removed, *ESC* only checks the $S2$ points in $D(s_1^2)$, instead of the entire $S2$ points. In this example, s_2^2 becomes a new $S1$ point, so it is removed from $S2$. We formally define a *dominance set* and establish Lemma 1 which states that a dominance set must contain all the necessary $S1$ candidate points as follows:

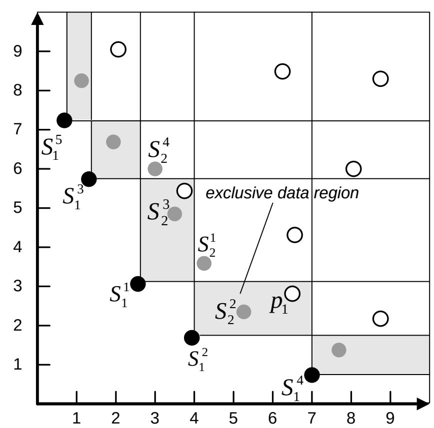


Fig. 3. Dominance set v.s. *EDR* set

Definition 2: (*Dominance Set:* $D(s_1^i)$)
A *dominance set* of a skyline point s_1^i (denoted by $D(s_1^i) = \{s_2^r, ...s_2^v\}$) is a $S2$ subset where $\forall s_2^w \in D(s_1^i), s_1^i \vdash s_2^w$, and $0 \leq (s_2^w.mindist - s_1^i.mindist) \leq (s_2^w.mindist - s_1^t.mindist)$, $\forall s_1^t \in (S1 - s_1^i)$. Each $D(s_1^i)$ is exclusive from any other dominance set; therefore, $S2 = D(S1)$, where $D(S1) = D(s_1^1) + ... + D(s_1^m)$ and m is the size of $S1$.

Lemma 1: Given a $D(s_1^i)$. Let A be the skyline points extracted from $EDR(s_1^i)$. $D(s_1^i)$ must contain A (A is a subset of D(s_1^i)).

Proof: (By contradiction) Let $p \in A$ be a point not included in $D(s_1^i)$. This is a contradiction, since p is only dominated by s_1^i. Therefore, it must be in $D(s_1^i)$. Therefore, $D(s_1^i)$ must contain all points in A.∎

In Figure 3, $D(s_1^2) = \{s_2^1, s_2^2\}$ contains two $S2$ points in the set which is a superset of $A = \{s_2^2\}$. One can observe that some non-exclusive $S2$ points (e.g., s_2^1 and s_2^4) can be assigned to different dominance sets. Intuitively, the $S1$ point with the minimal *mindist* to the query point (which has the largest dominance area) may contain the most $S2$ points. Thus, it might produce a load imbalance problem because the query processor needs to perform many dominance checks when a skyline point with a short *mindist* moves. To ensure that each dominance set has evenly distributed $S2$ points, the *ESC* algorithm inserts a non-exclusive $S2$ point s_2^w into $D(s_1^j)$, where s_1^j has the minimal value of $(s_2^w.minsit - s_1^j.mindist)$ among all other $S1$ points. In our algorithm, we utilize the *BBS* approach to initially compute the skyline query results. Along with the query evaluation, $S2$ points and the dominance set of each $S1$ point are computed during the execution of the modified *BBS* dominance-checking procedure which runs a window query to determine a set of candidate skyline points. Let e be the next discarded entry during the process of the dominance-checking procedure (e is dominated by some $S1$ point). Therefore, the algorithm proceeds to insert e into a dominance set and examine whether e is a $S2$ point. Given is a heap $H = \{s_1^i...s_1^k\}$ that is the set of the existing skyline points whose entries intersect with e. Since *BBS* always visits entries in the ascending order of their *mindist*, we have $\forall s \in H$, $s.mindist < e.mindist$. With the sorting of H by the *mindist* in descending order, $\exists s_1^j \in H$, $s_1^j \vdash e$ and the value of $(e.minsit - s_1^j.mindist) > 0$ is minimal among all other $S1$ points. Next, Lemma 2 is provided to prove the correctness of the $S2$ extraction.

Lemma 2: Given a point p which is dominated by $S1' = \{s_1^i...s_1^j\}$, where $S1' \subset S1$. If $\forall s_2^t \in D(S1'), s_2^t \nvdash p$, p must be a $S2$ point.

Proof: Since p is not dominated by $(S1 - S1')$, p can never be dominated by any $S2$ point in $D(S1 - S1')$ either, by transitivity. Therefore, if p is not dominated by any $S2$ point in $D(S1')$, p is guaranteed to be a final $S2$ point.∎

The pseudo code is shown in Algorithm 1, where the additional conditions (Lines 10-16 and 19-27) are inserted into the dominance-checking code for retrieving $S2$ points and determining the dominance sets. Line 4 sorts the heap in descending order of the *mindist* such that the skyline points with larger *mindist* are examined first. Line 12 obtains the dominating skyline point e_r for p which is inserted into $D(e_r)$ later. Based on Lemma 2, Lines 13-15 check whether p is a $S2$ point. Lines 20-23 ensure that each $S2$ is a data point. If e is an intermediate node, *BBS* is performed to retrieve local skyline points from the entry. Lines 23 and 25 insert the final $S2$ points O' into $S2$ and updates the $S2$ set by deleting those $S2$ points that are dominated by O'. To find such a set, the algorithm performs *S2Rtree.W*(O'), which is a window query that finds the $S2$ points in the dominance areas of O'.

3.3 Description of the ESC Algorithm

The main procedures of the *ESC* algorithm include *S1Evaluation* for the $S1$ updates and *S2Evaluation* for the $S2$ set maintenance. *ESC* delegates most of expensive computations that are irrelevant to $S1$ query results to *S2Evaluation*.

Algorithm 1 ESC dominance-check(p)

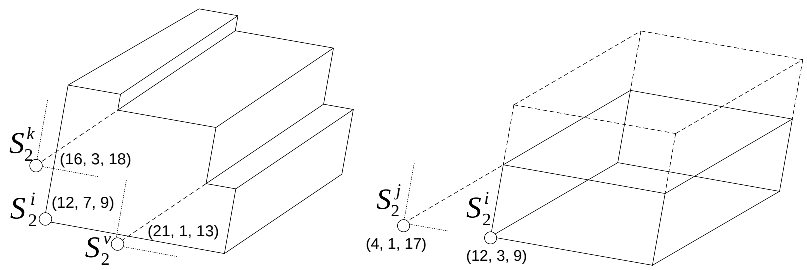
insert all entries of the root R in the heap
isDominated = false, $e_r = \phi$
while heap not empty **do**
  remove top heap entry e //the heap is sorted in descending order of *mindist*.
  if (e is an intermediate entry) **then**
    for (each child e_i of e) **do**
      if (e_i intersects with p) **then** insert e_i into heap
    end for
  else
    if ($e \vdash p$) **then**
      isDominated = true;
      let $e_r = e$, if e_r is not empty //e_r: the first $S1$ point dominating p
      for (each $S2$ skyline point $s_2^i \in D(e)$) **do**
        if ($s_2^i \vdash p$) **then** set p as a regular data point and return isDominated
      end for
    end if
  end if
end while
if (isDominated) **then**
  if (p is an intermediate entry) **then**
    perform DataRtree.BBS(p) that returns a skyline point set O
    let $O' \in O$ be the data set that is not dominated by $S2$.
    $S2 = S2 + O' - S2Rtree.W(O')$ and insert O' into $D(e_r)$
  else
    $S2 = S2 + p - S2Rtree.W(p)$ and insert p into $D(e_r)$
  end if
end if
return isDominated

To improve the performance of *S2Evaluation*, we introduce the concept of an *approximate exclusive data region* (*AEDR*) that helps to reduce the amortized cost of the $S2$ updates. When $d = 2$, the traditional *EDR* is a regular rectangle. However, a *EDR* has an irregular shape in higher dimensions. For example, in Figure 4(a), s_2^i is a skyline point to delete. The *EDR* is a irregular rectangle after deleting the overlapping area with the dominance area of s_2^k and s_2^v. Based on this observation, we can obtain a regular shaped *EDR* only when we consider the skyline points which have a value x^i larger than that of s_2^i in only one dimension. Because these points are completely "outside" of the *EDR*, they can trim the entire areas that represent the upper dimensional value x^i.

Definition 3: (*AEDR*)
Let $s_2^i = (x^1, x^2, ..., x^d)$, and $s_2^j = (y^1, y^2, ..., y^d)$. $AEDR(s_2^i) = s_2^i.DomArea - (s_2^i.DomArea \cap s_2^j.DomArea)$, $\forall s_2^j \in (S2 - s_2^i)$, there exists exactly one $x^k < y^k$, $1 \leq k \leq d$.

For example, in Figure 4(b), s_2^i is the skyline to delete and the solid rectangle box is an *AEDR*, which is a regular shape resulting from trimming the overlapping dominance areas of s_2^i and s_2^j. *ESC* utilizes the *AEDR* to search for the new $S2$ points by traversing the R-tree. Each *MBR* e extracted from the heap is checked whether it intersects with the *AEDR*. If true, *ESC* checks whether e is dominated by the existing $S2$ points.

When a $S1$ point p is newly inserted into the system or when it moves, *ESC* needs to re-group a new dominance set for p. A simple solution is to check every $S2$ point which currently belongs to a dominance set of some $S1$ point



(a) 3-*d* EDR example (b) AEDR example

Fig. 4. Traditional *EDR* v.s. *AEDR*

and migrate the $S2$ point to the dominance set of p if necessary. Instead, we provide *FindDomSet*, (the pseudo code is presented in Algorithm 2) applying the following Lemma that presents a heuristic to avoid checking the entire $S2$ set.

Lemma 3: Given a new $S1$ point s_1^k, re-group the points in $D(s_1^i)$, only where $\forall s_1^i \in (S1 - s_1^k)$, $s_1^i.mindist \leq s_1^k.mindist$.

Proof: Proof by definition. Let s_1^w be a $S1$ point that has the value of ($s_1^w.mindist > s_1^k.mindist$). $\forall p \in D(s_1^w)$, the value of ($p.mindist - s_1^w.mindist$) must be smaller than the value of ($p.mindist - s_1^k.mindist$). p must remain in the same dominance set of s_1^w. Therefore, it is not necessary to re-group these points in $D(s_1^w)$. ∎

Algorithm 2 *FindDomSet*(s_1^k)

for (each $p \in D(s_1^i)$, where $s_1^i \in (S1 - s_1^k)$ and $s_1^i.mindist < s_1^k.mindist$) **do**
  if ($s_1^k \vdash p$) **then**
    $D(s_1^i)$.remove(p)
    $D(s_1^k)$.insert(p)
  end if
end for

The *ESC* algorithm is implemented in an event-driven fashion to handle the skyline query updates. The main procedures include *S1Evaluation* (Algorithm 3) and *S2Evaluation* (Algorithm 4). When the query processor receives a request ($S1$, $S2$, or regular data point), it first performs *S1Evaluation* to examine whether the request affects the $S1$ set (the query result) and outputs the updated $S1$ points if the set has been modified. Then *S2Evaluation* processes the rest of non-$S1$-related computations. In the *S1Evaluation* procedure, Line 6 performs the *S1Rtree.dominace-descending* function where the dominance checks access the *S1Rtree* in the descending order of the *mindist* of the entries. We use the same principle of the *ESC dominance-check* algorithm (discussed in Section 3.2) to find the dominating $S1$ point s_1^k (Line 7) for a request point p. If p becomes a new $S2$ point evaluated by *S2Evaluation*, p is inserted into $D(s_1^k)$.

Lines 9−10 update the $S1$ set if p is a new $S1$ point and delete the I set, which is an existing $S1$ set dominated by p. I is obtained by executing a window query *S1Rtree.W*(p), using the dominance area of p as the range on the *S1Rtree*. Line 11 inserts the new $S1$ point p into $\widetilde{S1}$ and *S1Evaluation* will later pass this set to *S2Evaluation* where *FindDomSet*$(\widetilde{S1})$ is performed to find a $S2$ set for $D(p)$. Since all the points in I become new $S2$ points (inserted into $\widetilde{S2}$ in Line 12), the $S2$ set is updated later in *S2Evaluation* by adding the $\widetilde{S2}$ set. Lines 15−24 basically check all the $S2$ points $\in D(p')$ whether they are still dominated by p after p moves or is removed from the system. In Line 18, since o (new $S1$ point after p moves) can never dominate any $S1$ point, o is added to the $S1$ set directly. This is because o is an exclusive data point, and therefore o must not dominate any existing $S1$ points.

Algorithm 3 *S1Evaluation*(p)

1: let $\widetilde{S1} = \phi$ be a new $S1$ point set
2: let $\widetilde{S2} = \phi$ be a new $S2$ point set
3: let $\overline{S2} = \phi$ be the existing $S2$ points to remove
4: p' be the last-updated point of p
5: $S1 = S1 - p'$, if p was a $S1$ point
6: $isDomByS1 = S1Rtree.dominace\text{-}descending(p)$
7: let s_1^k be the $S1$ point with the minimal $(p.minsit - s_1^k.mindist)$ value among all other $S1$ points
8: **if** ($isDomByS1$ == false) **then**
9:     $I = S1Rtree.W(p)$
10:     $S1 = S1 + p - I$
11:     $\widetilde{S1}$.insert(p)
12:     $\widetilde{S2}$.insert(I)
13:     $D(p)$.insert(i), $\forall i \in I$
14: **end if**
15: **if** (p was a $S1$ point) **then**
16:     **for** (each $o \in D(p')$) **do**
17:         **if** ($S1Rtree.dominace\text{-}descending(o)$ == false) **then**
18:             $S1 = S1 + o$
19:             $D(p)$.remove(o)
20:             $\widetilde{S1}$.insert(o)
21:             $\overline{S2}$.insert(o)
22:         **end if**
23:     **end for**
24: **end if**
25: output the updated $S1$ set and continue *S2Evaluation*$(p, isDomByS1, s_1^k, \widetilde{S1}, \widetilde{S2}, \overline{S2})$ procedure

S2Evaluation is a more expensive procedure than *S12Evaluation*, because it involves *AEDR* computations to find a set of new $S2$ points to substitute a moving or removed $S2$ point. Lines 6−7 are processed if p is a new $S2$ point. The insertion of p may dominate some existing $S2$ points; therefore, Line 6 finds the dominated $S2$ points (*S2Rtree.W*(p)) and removes them from the $S2$ set. Similarly, in Line 10, since each point in $\widetilde{S2}$ was originally a $S1$ point, the $D(\widetilde{S2})$ set is directly removed from the $S2$ set without performing a window query to look for the dominated points. The deletion of the $S2$ point set $\overline{S2}$ is executed in Lines 11−12 and A contains the substitute $S2$ points, after $\overline{S2}$ is removed from the $S2$ set. Finally, *FindDomSet* is performed to find a group of $S2$ points for each point in $\widetilde{S1}$.

Algorithm 4 *S2Evaluation*(p, *isDomByS1*, s_1^k, $\widetilde{S1}$, $\widetilde{S2}$, $\overline{S2}$)

Let p' be the last-updated point of p
$\overline{S2}$.insert(p'), if p was a $S2$ point
if (*isDomByS1* == true)) **then**
  isDomByS2 = *S2Rtree.dominace*(p)
  if (*isDomByS2* == false)) **then**
    $S2 = S2 + p -$ *S2Rtree.W*(p)
    $D(s_1^k)$.insert(p) and $D(s_1^{k'})$.remove(p), where $s_1^{k'} (\neq s_1^k)$ was the dominating point of p
  end if
end if
$S2 = S2 + \widetilde{S2} - D(\widetilde{S2})$
A = *DataRtree-AEDR*($\overline{S2}$), where A is a regular data set and is not dominated by $S2$ points.
$S2 = S2 - \overline{S2} + A$ //A substitutes $\overline{S2}$
FindDomSet($\widetilde{S1}$)

4 Experimental Evaluation

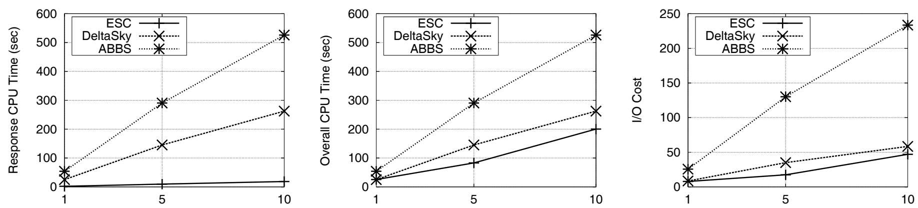
We evaluated the performance of the *ESC* algorithm by comparing it with the well-known *BBS* approach [14] and the *DeltaSky* algorithm [19]. For the *EDR* computations in BBS, we adopt the *ABBS* (Adaptive Branch-and-Bound Search) [19] to avoid complex irregular-shaped *EDR* computations. *ABBS* basically traverses the R-tree and determines whether an intermediate *MBR* e_i intersects with the dominance area of a skyline to delete. If this is true, it further checks whether any existing skyline dominates e_i. All of these algorithms utilize R-trees as the underlying structure for indexing the data and skyline points. We use the Spatial Index Library [7] for the R-tree index. A page size of 4Kbytes is deployed, resulting in node capacities between 94 ($d = 5$) and 204 ($d = 2$). $S1$ and $S2$ sets are indexed by a main-memory R-tree to improve the performance of the dominance checks. Our data sets are generated on a terrain service space of $[0, 1000]$ with the random walk mobility model [10]. Each object moves with a constant velocity until an expiration time. The velocity is then replaced by a new velocity with a new expiration time. We generated from 100,000 to 1,000,000 normal distributed data points with a dimension in the range of 2 to 5. The object update ratio is set in a range from 1% to 10%. Experiments are conducted with a Pentium 3.20 GHz CPU and 1 GByte of memory. The query results are evaluated in an event-driven approach. Therefore, the query processor calls different procedures based on each specific event type. The main measurement in the following simulations is the response CPU time (from receiving a data update request to the $S1$ update completion time or the evaluation time of *S1Evaluation*) and the overall CPU time (the evaluation time of *S1Evaluation* plus *S2Evaluation*). For *ABBS* and *DeltaSky* the overall CPU time also represents the response time. Our experiments use several metrics to compare these algorithms. Table 2 summarizes the default parameter settings in the following simulations.

Parameter	Default	Range
P	100,000	100,000, 500,000, 1,000,000
d	5	2, 3, 4, 5
f_{update}	10%	1%, 5%, 10%

Table 2. Simulation parameters

4.1 Update Ratio

First, we evaluated the impact of the update ratio. Figures 5(a) and (b) show the response time and overall CPU time as a function of update ratio, respectively, and Figure 5(c) illustrates the I/O cost for the three methods. We fix the data cardinality at 100,000 and dimensionality at 5. The *ESC* approach achieves a better performance than *ABBS* and *DeltaSky* for all update rates. The degradation of *DeltaSky* is caused by the expensive Maximum Coverage computations scanning over the projection lists and the increase of skyline point size which incurs bigger projection lists. *ESC* also outperforms both methods in terms of the overall CPU time, since the amortized cost of the *AEDR* computations and exclusive data evaluation is lower than the other two methods.

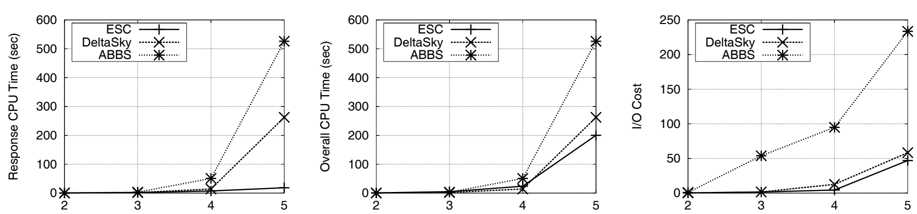


(a) Response CPU time (b) Overall CPU time (c) I/O cost

Fig. 5. Performance v.s. Update Ratio ($P = 100$k, $d = 5$)

4.2 Dimensionality

Next we report on the impact of the dimensionality on the performance of all three methods. Figures 6(a)(b)(c) show the CPU overheads and I/O cost v.s. the dimensionality ranging from $d = 2$ to 5, respectively. When d increases, the performance of all methods is degraded because the exclusive data point computations are complex and R-trees fail to filter out irrelevant data entries in higher dimensions. From all the figures, we can see that *ESC* outperforms *ABBS* and *DeltaSky* in terms of the CPU time and I/O cost.

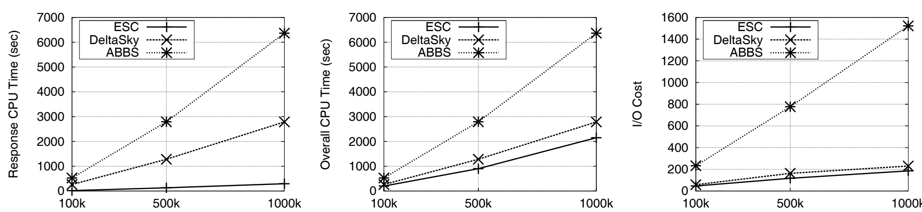


(a) Response CPU time (b) Overall CPU time (c) I/O cost

Fig. 6. Performance v.s. Dimensionality ($P = 100$k, $f_{update} = 10\%$)

4.3 Cardinality

Figures 7(a)(b) show the response and overall CPU time as a function of the number of data points, respectively, and Figure 7 (c) illustrates the corresponding I/O cost. Overall, the CPU overheads increase as a function of the number of data points. *ESC* achieves a significant reduction in terms of the response CPU time compared to *ABBS* and *DeltaSky*. *ESC* takes advantage of the pre-computed $S2$ points retrieved by the latest *S2Evaluation* procedure and quickly locates relevant new $S1$ candidates for substituting a removed or moving $S1$ point. As we can see from the experimental results, the adoption of *AEDR* helps *ESC* to achieve better overall CPU performance and competitive I/O cost with *DeltaSky*.



(a) Response CPU time (b) Overall CPU time (c) I/O cost

Fig. 7. Performance v.s. Cardinality ($d = 5$, $f_{update} = 10\%$)

5 Conclusions

In this paper, we propose an incremental skyline update approach. Our *ESC* algorithm achieves a faster response time and overall CPU performance. With the adoption of the pre-computed $S2$ sets, *ESC* can efficiently update the skyline query results and delegate the most complex computations to a separate procedure that executes after the updates of the query results are completed. An approximate exclusive data region (*AEDR*) is proposed and our experiments confirm the feasibility of *AEDR* which has a low amortized cost of the exclusive data evaluation in high dimensional and dynamic data environments. The *S1Evaluation* procedure first examines all the incoming data requests and updates the $S1$ result if necessary and the *S2Evaluation* procedure integrates our lemmas and heuristics to achieve a low CPU overhead and reduced I/O cost.

References

1. S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering (ICDE), Heidelberg, Germany*, pages 421–430, 2001.
2. C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA*, pages 203–214, 2005.

3. V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.
4. Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous Skyline Queries for Moving Objects. *IEEE Trans. Knowl. Data Eng.*, 18(12):1645–1658, 2006.
5. D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China*, pages 275–286, 2002.
6. K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the Skyline in Z Order. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), University of Vienna, Austria*, pages 279–290, 2007.
7. S. I. Library. http://www.research.att.com/ marioh/spatialindex/index.html.
8. X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *Proceedings of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan*, pages 502–513, 2005.
9. X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting Stars: The k Most Representative Skyline Operator. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE), Istanbul, Turkey*, pages 86–95, 2007.
10. A. B. McDonald. A mobility-based framework for adaptive dynamic cluster-based hybrid routing in wireless ad-hoc networks. *Ph.D. Dissertation proposal, University of Pittsburgh, 1999.*, 1999.
11. M. D. Morse, J. M. Patel, and W. I. Grosky. Efficient Continuous Skyline Computation. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE), Atlanta, GA, USA*, page 108, 2006.
12. M. D. Morse, J. M. Patel, and H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), University of Vienna, Austria*, pages 267–278, 2007.
13. D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478, New York, NY, USA, 2003.
14. D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.
15. J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), Trondheim, Norway*, pages 253–264, 2005.
16. M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), Seoul, Korea*, pages 751–762, 2006.
17. K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 301–310, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
18. L. Tian, L. Wang, P. Zou, Y. Jia, and A. Li. Continuous Monitoring of Skyline Query over Highly Dynamic Moving Objects. In *Sixth ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), Beijing, China*, pages 59–66, 2007.
19. P. Wu, D. Agrawal, Ö. Egecioglu, and A. E. Abbadi. Deltasky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE), The Marmara Hotel, Istanbul, Turkey*, pages 486–495, 2007.