Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 3411–3437
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Efficient continuous skyline computation

M. Morse [a,*], J.M. Patel [a], W.I. Grosky [b]

[a] Department of Electrical Engineering and Computer Science, University of Michigan–Ann Arbor, Ann Arbor, MI 48105, USA
[b] Department of Computer and Information Science, University of Michigan–Dearborn, Dearborn, MI 48128, USA

Received 22 September 2006; received in revised form 22 February 2007; accepted 28 February 2007

Abstract

In a number of emerging streaming applications, the data values that are produced have an associated time interval for which they are *valid*. A useful computation over such streaming data is to produce a continuous and valid *skyline* summary. Previous work on skyline algorithms have only focused on evaluating skylines over static data sets, and there are no known algorithms for skyline computation in the continuous setting. In this paper, we introduce the *continuous time-interval skyline* operator, which continuously computes the current skyline over a data stream. We present a new algorithm called *LookOut* for evaluating such queries efficiently, and empirically demonstrate the scalability of this algorithm. In addition, we also examine the effect of the underlying spatial index structure when evaluating skylines. Whereas previous work on skyline computations have only considered using the R^*-tree index structure, we show that for skyline computations using an underlying quadtree has significant performance benefits over an R^*-tree index.

Keywords: Database; Skyline; Algorithm; Index; Multidimensional data structures; Quadtree; R^*-tree

1. Introduction

The skyline operator is an elegant summary method over multidimensional datasets [14]. Given a dataset P containing data points $p_1, p_2, \ldots, p_n$, the skyline of P is the set of all p_i in P such that no p_j *dominates* p_i. A commonly cited example for the use of a skyline operator is assisting a tourist in choosing a set of *interesting* hotels from a larger set of candidate hotels. Each hotel is identified by two attributes: a distance from a specific point (such as a location on a beach), and the price for the hotel. To assist a tourist in narrowing down the choices, the skyline operator can be used to find the set of all hotels that are not dominated by another hotel. Hotel *a dominates* hotel *b* if *a* is at least as close as *b* and at least as cheap as *b*, and offers either a better price, or is closer, or both compared to *b*. Fig. 1 shows an example dataset and the corresponding skyline; the distance of the hotel from the beach is shown on the *x*-axis and the hotel price is plotted along the *y*-axis. The skyline is the set of points *a, c, d, i*, and *j*.

* Corresponding author.
E-mail addresses: mmorse@eecs.umich.edu (M. Morse), jignesh@eecs.umich.edu (J.M. Patel), wgrosky@umich.edu (W.I. Grosky).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.02.033

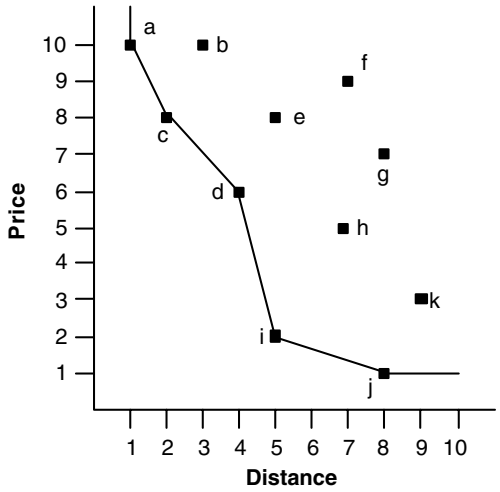


Fig. 1. Example dataset and its skyline.

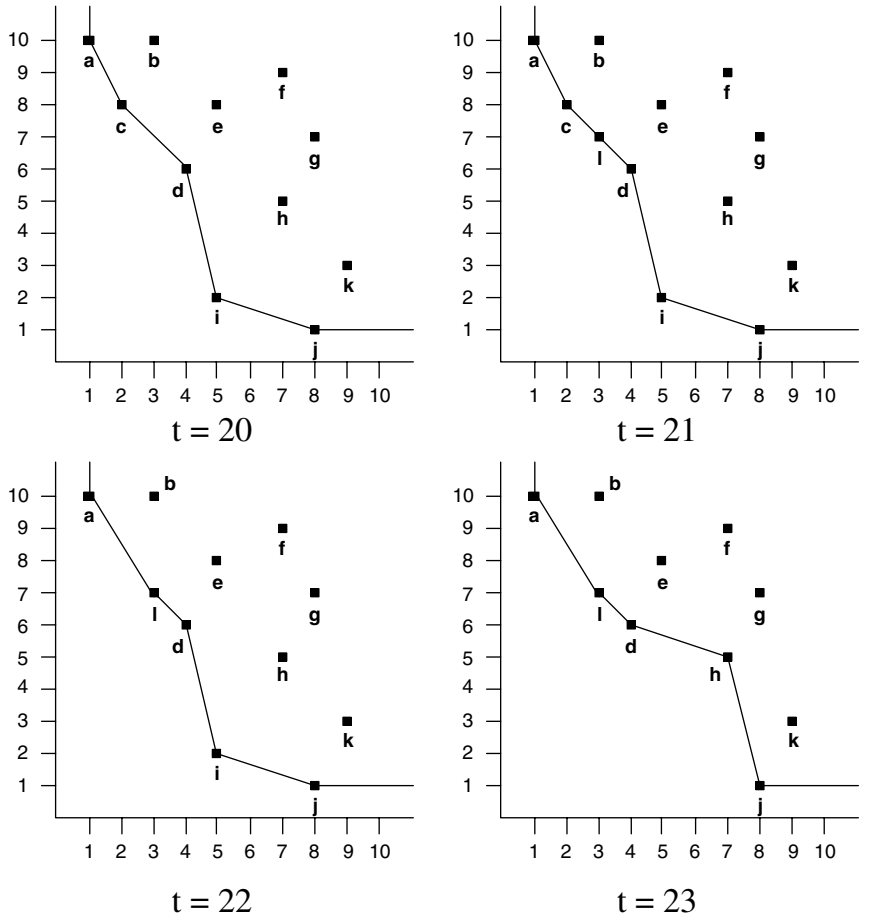
The skyline can be generalized to multidimensional space where a point a dominates another point b if it is as good or better than b in all dimensions, and is better than b in at least one dimension. Implicit in this definition of the skyline operation is the notion of comparing the *goodness* along each dimension. A common function for determining this property is to use the *min* function. However, skyline computation can easily be extended to consider other functions, such as *max*.

Skyline algorithms to-date assume that the dataset is static, i.e. the data has no temporal element associated with it, or have dealt with temporal data only in a sliding window context, i.e. the skyline is evaluated only over the most recent n data points. In contrast, the *continuous time-interval skyline* operation involves data points that are continually being added or removed. Each data point has an arrival time and an expiration time associated with it that defines a time interval for which the point is valid. The task for the database then is to *continuously* compute a skyline for the data points that are valid at any given time. The continuous time-interval model used in this paper is a more general one than the sliding window used in [34,17], and hence the techniques discussed in this paper may also be used to evaluate such sliding window queries.

Fig. 2 shows the difference between a conventional skyline query such as that seen in Fig. 1 and a continuous time-interval skyline over a similar dataset. Each data point has an arrival time and an expiration time, as shown in the table on the right hand side of the figure. The figure displays the skyline as it transitions from time 20 to 23. At time 20, the skyline is the same as that in Fig. 1. The skyline changes at time 21 when data point l arrives. It is part of the new skyline. At time 22, c expires, and the skyline must be modified to remove c from both the dataset and the skyline. Notice that b is not in the new skyline, since b is dominated by both a and l. At time 23, data point i expires, and the skyline is modified again, this time introducing a new point into the skyline, point h.

There are a number of emerging streaming applications that require efficient evaluation of the *continuous time-interval skyline*. If we consider the familiar example of choosing hotels, hoteliers routinely run competitive deals with booking agencies such as priceline.com. These hotel operators may wish to submit a bid for their rooms at a particular price for some specified period of time. If bookings increase, they may wish to increase the room cost, or conversely decrease it if bookings do not increase. A user interface on top of the raw priceline data may wish to show the most competitive rooms (with respect to the beach for a given price) to customers, while balancing bids from many hotel companies that all may change with time. At any given time, there may be many continuous skyline queries active in the system, depending on a number of other user preferences (such as distance from a customer-specific point of interest). In such a case, the server needs to efficiently evaluate a large number of skyline queries continuously on data points with arbitrary valid time ranges. Such an application could be useful for online hotel bookers, such as orbitz.com [1].

Another example for the use of continuous skyline evaluation is in the realm of online stock trading. Traders are interested not only in the trading price of a stock, but also in the number of shares trading hands at a



p	t_a	t_e
a	1	29
b	3	27
c	4	**22**
d	6	24
e	8	31
f	14	33
g	16	50
h	17	30
i	18	**23**
j	19	45
k	20	40
l	**21**	37

Fig. 2. The example data with arrival and expiration times. The continuous skyline is shown in transition from time 20 to 23.

price. Since trades are temporal, traders may only be interested in trades within the last hour. Hence, a mechanism for allowing trades to age out of the system after an expiration time is needed. In such a scenario, traders are interested in the skyline (price versus share volume) for many different stocks. Each stock may require a different continuous time-interval skyline operator to keep track of the latest developments. Note that in such applications there can be a large number of skyline queries that the server may need to evaluate continuously, which demand time and space efficient evaluation methods.

In this paper we present the first algorithm for efficiently evaluating the continuous time-interval skyline operation. We show that this new algorithm, called *LookOut*, is very scalable as it is both time and space efficient. *LookOut outperforms an iterative algorithm based on currently known methods by at least an order of magnitude in most cases!* We also compare *Lookout* with the *lazy* and *eager* methods of [34], and show that it performs better than either of these methods for anti-correlated datasets while evaluating a more general time model than the sliding window queries.

The other contribution that we make in this paper is to explore the choice of index structures for evaluating skyline operations (both in the static and the continuous cases). All previous skyline algorithms that have used spatial indices have employed the *R*-tree family of indices [9]. For example, the branch and bound algorithm (*BBS*) [24,25] uses the R^*-tree index [2]. We make an important observation that the MBR overlap involved with the R^*-tree's partitioning dramatically increases the number of both index non-leaf and leaf nodes that are examined during a skyline evaluation. In contrast, the non-overlapping partitioning of a quadtree is far superior for computing skylines.

We note that an immediate question that arises with a quadtree index is that it is not a balanced indexing structure. However, it has been shown to be an effective disk-based index [8,11] and some commercial object-relational systems already support quadtrees [15]. The claim that we make and support is that if the speed of skyline computation is critical, a quadtree is far more preferable than an R^*-tree. In our skyline experiments, *the quadtree index significantly speeds up skyline computation by up to an order of magnitude or more in some cases, and is never slower than the R^*-tree approach*. Using the quadtree also results in smaller memory consumption during the skyline computation. We note that the issue of comparing the R-tree and quadtree for a wider range of spatial operations is beyond the scope of this paper. Our results show that in systems that support quadtrees, using them is preferable for skyline computation.

It is also worth mentioning that the time-interval model that we use in this paper is very flexible, and can easily accommodate more specialized streaming data models. For example, our model can be used with datasets that have no expiration time by setting the expiration time of the data in the model to infinity. Similarly, pre-existing data or data that does not have any implicit start time, can simply be treated as having a start time of zero. In addition, data that does not have an explicit expiration time, but rather is valid for t seconds from its arrival can simply be handled by noting its arrival time, a, and setting its expiration time to $t + a$.

The remainder of this paper is organized as follows: related work is covered in Section 2, and we present our new algorithm in Section 3. In Section 4, we consider the effect of the indexing structure for skyline computation. Experimental evaluations are presented in Section 5, and finally Section 6 contains our conclusions.

2. Related work

The skyline query is also referred to as the Pareto curve [26] or a maximum vector [16]. The skyline query is related to several other well-known problems that have been studied in the literature. Nearest neighbor queries were proposed by [28] and studied in [10], top-N were studied in [6], the contour problem in [22], convex hulls in [27,3], multidimensional indexing [19,30,35], and multi-objective optimization in [31,26].

The skyline algorithm was first proposed by Kung et al. [16], which employs a divide-and-conquer approach. Borzsonyi et al. [4] introduced the *skyline* operation in a database context and showed how the standard indexing structures, B-trees and R-trees, could evaluate skyline queries. Chomicki et al. [7] formulated a generic relational-based approach to compute the skyline, based on the approach of [4]. An algorithm for high dimensional skyline computation was proposed by Matoušek [21], and a parallel algorithm was proposed by Stojmenovic and Miyakawa [32].

An algorithm to obtain the skyline based on a nearest neighbors approach was introduced by Kossmann et al. [14], which uses a divide-and-conquer scheme for data indexed by an R-tree. Two algorithms were proposed in [33]. One is a bit mapped approach, and the other is an indexed approach using B-trees.

The branch and bound technique for skyline computation (*BBS*) was proposed by Papadias et al. in [24,25]. It traverses an R^*-tree using a best-first search paradigm, and has been shown to be optimal with respect to R^*-tree page accesses. Currently, *BBS* is the most efficient skyline computation method, and in this paper we compare the *LookOut* algorithm with *BBS*. *BBS* operates by inserting entries into a heap ordered by a specified distance function. At each stage, the top heap entry is removed. If it is a R^*-tree node, its children are inserted into the heap. If it is a point, it is tested for dominance by other elements of the growing skyline and is either discarded or added to the skyline. This algorithm requires $O(s \cdot \log(N))$ R^*-tree page accesses, where s is the number of skyline points and N is the dataset cardinality. Papadias et al. [25] also discusses skyline maintenance in the presence of explicit updates, but does not discuss time-interval skylines on streams.

Lin et al. [17] focus on computing the skyline against the most recent n of N elements in a data stream. Their approach indexes data in an R-tree and uses an interval tree to determine when a point is no longer amongst the most recent N points. They also propose a continuous skyline algorithm based around the n of N model which, similar to our algorithm, incorporates a heap to remove elements that have slipped outside the working window. But the similarities to our work end here. The window of size n necessitates a limited scope of elements in the dataset and thus in the skyline as well. Consequently, there is not an explicit temporal element to the computation of the skyline. In the temporal case, which we use in this paper, the number of points under consideration is *not* restricted by any N, and at any given point in time new points may arrive, old points may

expire, or any combination of the two. Consequently, with our model, the technique proposed in [17] cannot be directly applied. Data reduction in streaming environments is studied in [18].

Tao and Papadias [34] also studied sliding window skylines, focusing on data streaming environments. Their work also focuses on the most recent n window of data points. This is the most similar of the previous work to our work in this paper, and we compare the performance of the two techniques, *eager* and *lazy* proposed in the paper with *LookOut*.

Huang et al. [12] studies continuous skyline queries for dynamic datasets. Here, the data is moving in one or more dimensions. To efficiently evaluate continuous skyline queries in the presence of moving data, a kinetic-based data structure is developed. While this work is similar to our work because it requires the continuous evaluation of the skyline as the data changes, the data elements are moving as opposed to arriving at and expiring from the dataset. Since the data model of [34] is closer to our model, we compare *LookOut* with its *eager* and *lazy* techniques.

This paper is a full-length version of the short poster paper [23].

2.1. BBS example

We present the operation of *BBS* on the dataset shown in Fig. 3. This dataset consists of 6 data points indexed by an R^*-tree. Let us assume that each internal R^*-tree node can hold up to three entries, and that each leaf node can also hold up to three entries.

The *BBS* algorithm begins by inserting $R1$ into the heap that is ordered by the minimum Manhattan distance. The contents of the heap at each stage of the algorithm are shown in Table 1. $R1$ is popped off the heap and its children, $R2$ and $R3$, are inserted back into the heap. $R2$ has a Manhattan distance of 3, whereas $R3$ has a distance of 6, so $R2$ is popped off the heap and expanded first. The two children that compose its local

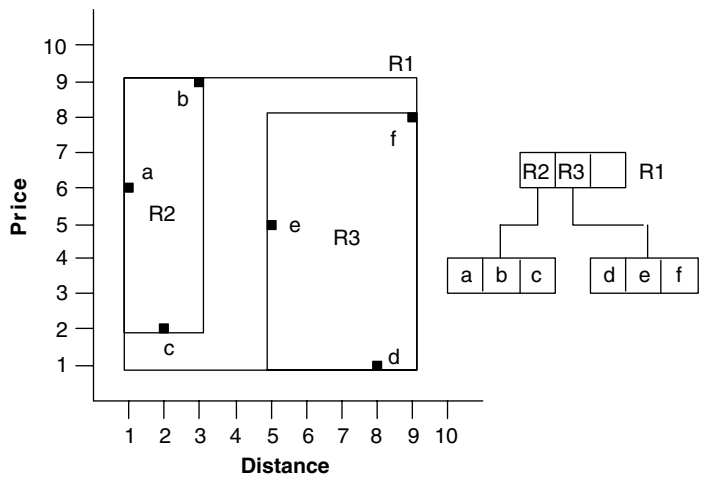


Fig. 3. A sample dataset indexed by an R-tree used to illustrate the operation of the *BBS* algorithm.

Table 1
Contents of the heap during an iteration of the *BBS* algorithm for the example dataset shown in Fig. 3

Action	Heap contents	(Skyline)
Expand $R1$	($R2$, 3), ($R3$, 6)	$\emptyset$
Expand $R2$	(c, 4), ($R3$, 6), (a, 7)	$\emptyset$
Add c	($R3$, 6), (a, 7)	$\{c\}$
Expand $R3$	(a, 7), (d, 9)	$\{c\}$
Add a	(d, 9)	$\{a,c\}$
Add d	Empty	$\{a,c,d\}$

skyline, c and a, are inserted back into the heap. Note that b need not be inserted back into the heap, since it is dominated by c. Since c is now at the top of the heap, it is popped off and inserted into the set of skyline points. Next, $R3$ is expanded. Two of its children, e and f, are not inserted back into the heap; e is dominated by c, and f is not part of the local skyline of $R3$. The heap now contains a and d. They are both popped off the heap and inserted into the skyline. The heap is now empty, and the *BBS* algorithm terminates.

3. The lookout algorithm

In this section, we present our algorithm for efficiently evaluating time-interval continuous skyline queries.

3.1. Overview

Each data point in the dataset is associated with an interval of time for which it is valid. The interval consists of the arrival time of the point and an expiration time for the point. The notation for the interval is (t_a, t_e).

The skyline in the continuous case may change based on one of two events: namely, (a) some existing data point i in the skyline may expire, or (b) a new data point j may be introduced into the dataset.

Algorithm 1. LookOut

[1]
Input: Index *Tree*, Heap *tHeap*, Current Time *Time* List *Skyline*, Set *DSP*,
Set *NSP*, Time *End*
Time < *End*
ndp is a new data point insert *ndp* into *Tree* insert *ndp* and expiration time into *tHeap*
isSkyline(*Tree*, *ndp*)
remove points from *Skyline* dominated by *ndp*
add *ndp* to *Skyline*
tHeap.top.expireTime equals *Time* delete *tHeap.top* from *TreetHeap.top* is a skyline
point add *tHeap.top* to *DSP*
point ∈ *DSP*
NSP ← MINI(*point*, *tree*)
t ∈ *NSP* if *isSkyline*(*Tree*, *t*) is *true*, add *t* to *Skyline*
update *Time* to the current time.

In the case of an expiration, the dataset must be checked for new skyline points that may have previously been dominated by i. These points must then be added to the skyline if they are not dominated by some other existing skyline points. In the case of insertion, the skyline must be checked to see if j is dominated by a point already in the skyline. If not, j must be added to the skyline and existing skyline points checked to see if they are dominated by j. If so, they must be removed.

The *LookOut* algorithm takes advantage of these observations to evaluate the time-interval continuous skyline. Since the skyline can change only when either a new point arrives or an old point expires, *LookOut* maintains the current skyline S. A data point p is inserted into a spatial index at time t_a. This point is checked to see if it is in the skyline, and if so, S is updated. If p is dominated, no changes are made to S. When t_e arrives, p is removed from the dataset and deleted from the spatial index. At this time, the dataset is checked to see if any of the points dominated by p are now elements of the skyline. If so, these points are added to S.

LookOut takes advantage of two important properties of hierarchical spatial indices, such as the R-tree family of indices and the quadtree.

(1) If p dominates the all corners of a node o (and hence dominates the entire region bounded by the node), then p dominates all of the points contained in o and its children.
(2) If all of the corners of a node o dominates a point p (and hence the entire region bounded by the node dominates p), then all of the points contained in o and its children dominate p.

These two observations are later used to prune nodes of the index and to discard new points from skyline consideration by *LookOut*.

Algorithm 2. IsSkyline

[1]
Input: Point P_{new}, Index node *Tree*
insert *Tree* into a heap *BHeap*, with distance 0.
BHeap isn't empty
Tree ← pop of *BHeap*
Tree is a leaf node
check if one of the entries of *Tree* dominates P_{new}
if so, return false. Otherwise, continue
Child ∈ the non-empty children of *Tree*
minimum corner of *Child* does not dominate P_{new}
continue
maximum corner of *Child* dominates P_{new}
return false
insert *Child* into *BHeap*
return true

3.2. Algorithm description

LookOut may be used with any underlying data-partitioning scheme. In our implementation, we chose to use and evaluate both the *R*-tree [2] and a disk-based PR quadtree [29]. We use the *R*-tree because of its ubiquity in multidimensional indexing and its use in other static-data skyline algorithms such as [24]. The quadtree index uses a regular non-overlapping partitioning of the underlying space, and is more effective in pruning portions of the index that need not be traversed for skyline computation (a discussion of these tradeoffs is presented in Section 4).

The *LookOut* algorithm is presented in Algorithm 1. As seen in line 4, when a new data point arrives, *LookOut* first stores the item into the spatial index. Each data element is also inserted into a binary heap (line 6) that is ordered on the expiration time. This heap is used so that data can be removed from the system when it expires. The element is then checked to see if it is a skyline point by the *isSkyline* algorithm (line 7), which will be explained shortly. If so, the new point is added to the skyline and those skyline points it dominates are removed. As time passes, the minimum entry in the binary heap is checked to see if its expiration time has arrived (line 12) and, if it has, it is deleted from the index. The skyline points themselves are maintained in a list, so that they may be returned immediately in the event of a skyline query over the dataset. (The skyline points can also be stored in an index, but the skyline is small in size and the index overhead often mitigates the benefits of using the index.) A separate heap, ordered on the expiration time, is also maintained for the skyline points so that an expired skyline point may be quickly removed. Those points that have been removed from the skyline (line 18) leave possible gaps that need to be filled by currently available data. The *MINI* algorithm finds the mini-skyline of points that were dominated by a deleted skyline point and effectively plugs a hole left by a deleted skyline point. Some and possibly all of the points found by *MINI* may be dominated by some other skyline point. Before adding them to the skyline, *LookOut* tests if each is in fact a new skyline point with *isSkyline* (line 21).

Algorithm 3. MINI

[1]
Input: Point P_{sky}, Index node *Tree*
Output: skyline *miniSkyline*
insert *Tree* into heap *BHeap*, with distance 0.

BHeap isn't empty
BHeap.top is a point
point ← pop *BHeap*
pIsDominated ← *FALSE*
each element *a* in *miniSkyline*
a dominates *point*
pIsDominated ← *TRUE*
pIsDominated is *FALSE*
insert *point* into *miniSkyline*
Tree ← pop of *BHeap*
P_{sky} dominates the maximum corner of *Tree*
Tree is a leaf node
find the local skyline of just *Tree*
point ∈ the local skyline of *Tree*
P_{sky} dominates *point*
insert *point* into *BHeap*.
Child ∈ the non-empty children of *Tree*
insert *Child* and *Child*'s distance into *BHeap*

The *isSkyline* algorithm is shown in Algorithm 2. It uses a best-first search paradigm, which is also used in *BBS*. The index nodes are inserted into a heap based on distance from the origin. When expanding a node in the heap (line 4 of Algorithm 2), the *isSkyline* algorithm discards any child node *n* whose lower left corner does not dominate P_{new} (line 10). This is because any data point in any such *n* cannot possibly dominate P_{new}, so for the purposes of skyline testing, it can be discarded. If the upper right corner of the child node (which isn't empty) dominates P_{new}, the algorithm can terminate and answer *false* (line 14). If the node is a leaf (line 5), the elements are compared against P_{new} for dominance. If any such element dominates P_{new}, the algorithm terminates and answers *false*. If the heap ordered on the minimum distance from the origin is ever empty, the algorithm answers *true*.

MINI, seen in Algorithm 3, is also a best-first search algorithm and maintains a binary heap. It takes as input a deleted skyline point P_{old}, which must dominate all points under consideration. It operates by popping the top element off the heap and inserting its children back into the heap, provided they are not dominated by the growing skyline. It has the extra caveat that all elements it inserts into the heap must be dominated by P_{old}. The algorithm begins by checking if the top heap element is a point (line 5). If the point is dominated by the growing mini skyline, it is ignored; else, it is added to the mini skyline (lines 8–15). If the upper right corner of any internal node is not dominated by P_{old}, then it may be discarded (line 18). If the top of the heap is a leaf (line 19), its local skyline is added to the heap (line 23). If the top is an internal node, those elements which have their upper right corner dominated by P_{old} are inserted back into the heap. *MINI* terminates when the heap is empty.

3.3. Example

We now consider an example execution of the *LookOut* algorithm on the dataset shown in Fig. 2. Fig. 2 depicts the example dataset beginning at time 20; at this time, the data points in an *R*-tree might resemble Fig. 4. Let us assume that each internal *R*-tree node can hold up to three entries, and that each leaf can also hold three. When *l* arrives at time 21, the *isSkyline* algorithm is run to determine if *l* is a skyline point. First, the root node of the *R* tree is accessed, and *R*5 is inserted into the heap, with a Manhattan distance (from the origin) of 6. *R*6 is not placed into the heap because its lower left corner does not dominate *l*. Thus, it may be ignored for the purposes of *isSkyline*. The top of the heap is then popped and processed. *R*5 contains two child nodes, *R*1 and *R*2, but since the lower left corner of neither of these dominates *l*, they are both discarded. Since the heap is empty, *l* is added to the skyline. The new node must also be inserted into the *R*-tree as well.

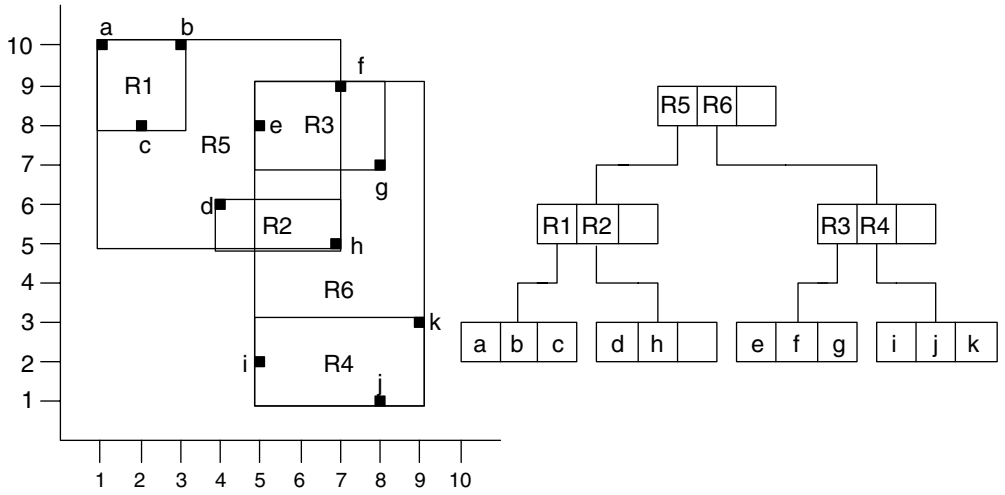


Fig. 4. An *R*-tree depicting the dataset in Fig. 1.

At time 22, *c* expires, and must be removed from the dataset. Following its removal from the index, the *R*-tree appears as shown in Fig. 5. The heap element identifies *c* as a skyline point. Since *c* is a skyline point, its removal may mean that pre-existing data points must be added to the skyline, so the *MINI* algorithm is run. The contents of *MINI*'s heap are depicted in Table 2. *MINI* begins by accessing the *R*-tree root. It adds *R*5 to its heap along with its Manhattan distance (6) and *R*6 with its Manhattan distance (6). Node *R*5 is removed and expanded; the only child of *R*5 that is added to the heap is *R*1, since the upper right corner of *R*2 is not dominated by *c*. *R*6 is next expanded, since its Manhattan distance is the smallest of any point or node in the heap, and *R*3 is added with a distance of 12. Next, *R*1 is expanded and *b* is added to the heap. None of the other children of *R*1 are added since they are not dominated by *c*. *R*3 is expanded and *e* and *f* are added to the heap. This ultimately produces *b* and *e* as the *MINI* skyline for entry *c*. Note that *f* is not included, since it is dominated by *e*. The *isSkyline* algorithm must now be called for both *b* and *e* to test if they are in fact skyline points. Neither one is; *e* is dominated by *d* and *b* is dominated by *l*. Therefore, no skyline change is required with the deletion of *c*.

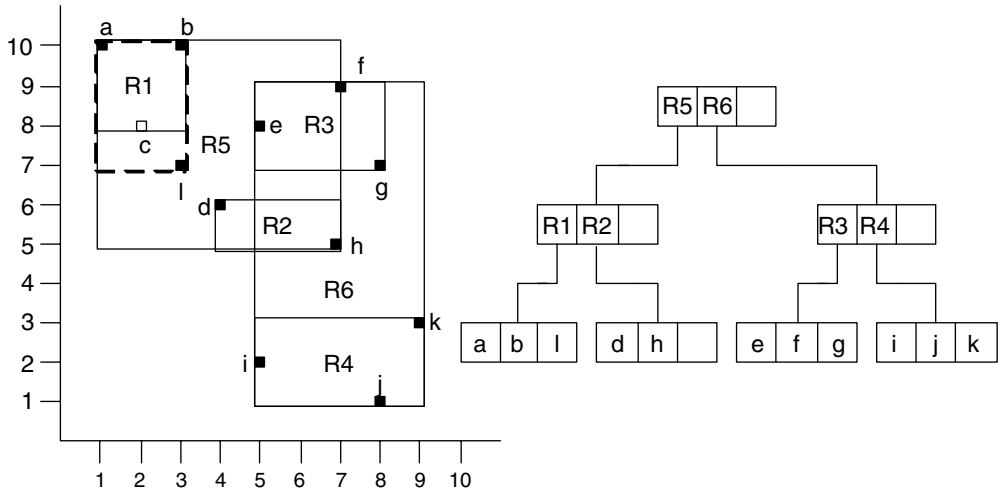


Fig. 5. An *R*-tree following the changes made to the dataset in Fig. 2 up to time 22.

Table 2
Contents of the heap during an iteration of the *MINI* algorithm

Action	Heap contents	(MINI Skyline)
Access root	($R5$, 6), ($R6$, 6)	$\emptyset$
Expand $R5$	($R6$, 6), ($R1$, 8)	$\emptyset$
Expand $R6$	($R1$, 8), ($R3$, 12)	$\emptyset$
Expand $R1$	($R3$, 12), (b, 13)	$\emptyset$
Expand $R3$	(b, 13), (e, 13), (f, 16)	$\emptyset$
Add b	(e, 13), (f, 16)	$\{b\}$
Add e	(f, 16)	$\{b,e\}$
Remove f	Empty	$\{b,e\}$

3.4. Analysis of LookOut in comparison to BBS

In this section, we examine the quantitative cost of *LookOut* and compare it against the cost of an iteration of the *BBS* algorithm. Note that since there are no current algorithms for continuous skyline evaluation, repeatedly running *BBS* can be considered to be the best alternative to *LookOut*. We observe that the only operations that can affect the skyline (and hence the cost of *LookOut*), are either an insert operation or a delete operation. During time intervals when one of these two operations do not occur, the skyline remains the same and *LookOut* performs no work. During this analysis, we consider indexing with an *R*-tree.

To determine the cost of an insertion, the costs of several operations need to be evaluated. These operations are: (a) the cost of adding an entry to the expiration-time heap, (b) the cost of adding an entry to the indexing structure, and (c) the cost of running the *isSkyline* algorithm, to determine if the new point is in the skyline.

The costs of both adding an entry to the heap and of inserting an entry into an index structure are identical for both *LookOut* and *BBS*. Consequently, neither one of these operations make *LookOut* perform either better or worse than *BBS*. The real difference between the two lies in the cost savings that *isSkyline* obtains over *BBS*.

First, we consider the worst case cost of *BBS* relative to the worst case cost of *isSkyline* for a single insertion operation. For *BBS*, the worst case occurs if all data points are in the skyline. In this case, all of the leaf and non-leaf nodes of the *R*-tree are inserted into the heap that *BBS* uses to order elements based on their minimum $L1$-norm distances. Each data point is also inserted into this heap when their respective leaf nodes are expanded. Since *BBS* checks each element removed from the heap relative to the growing skyline, this worst case cost is $O(n^2)$.

The worst case for *isSkyline* for a single insertion occurs if the new data point p that has been inserted overlaps with all leaf and non-leaf nodes of the *R*-tree. If this occurs, all of the leaf and non-leaf nodes of the *R*-tree are inserted into the heap ordered on the minimum distances to the origin. Each non-leaf or leaf node is inserted into the heap only once, based on the distance of the node from the origin. For example, the root node of the tree is inserted into the heap, and then expanded. Its children are then inserted into the heap. The root node is never considered by the algorithm again. Each of the nodes in the heap are expanded exactly once and only once, resulting in their children being inserted into the heap. When each leaf node is expanded, the entries in it are compared with the new data point. Each non-leaf or leaf node and each data point are compared in the worst case at most once with the new data point p. In the worst case, all of the entries in the dataset are compared with this new point, producing a worst case cost of $O(n)$ comparisons. For example, consider the case when all n data points are elements of the skyline and the new data point overlaps with the leaf level nodes of the tree that contain these points. Then, to determine if the new data point is in the skyline, all n nodes in the dataset will be compared with the new data point.

Second, we compare the average case cost of *BBS* to the average case cost of *isSkyline*. Since *isSkyline* only tests whether a single point is dominated by an existing point or not, whereas *BBS* computes an entire skyline from scratch, the cost savings is dependant on the number of elements in the skyline that *BBS* evaluates. If this number of skyline points is s, then the average case cost of *isSkyline* relative to that of *BBS* is approximately $1/s$.

To determine the cost of a single deletion operation, the costs of the following operations must be evaluated: (a) the cost of removing an entry from the expiration-time heap, (b) the cost of removing an entry from the indexing structure, and (c) the cost of running the *MINI* algorithm, to determine if alternate points must be added to the skyline.

The costs of both removing an entry from the heap and of removing an entry from the index structure are identical costs, regardless of whether *LookOut* or *BBS* is computing the skyline. Consequently, neither one of these operations make *LookOut* perform either better or worse than *BBS*. The real difference between the two for a deletion lies in the cost savings that *MINI* obtains over *BBS*.

Next, we consider the worst case cost of *BBS* relative to the worst case cost of *MINI* for a single deletion. For *BBS*, the worst case for a deletion is the same as it was in the case of an insertion and occurs if all data points are in the skyline. This worst case cost is $O(n^2)$.

The worst case cost for *MINI* occurs if the deleted data point is the only element in the skyline. In this case, *MINI* must evaluate a completely new skyline from scratch. The worst case for *MINI* then is the same as the worst case cost of *BBS*, which is $O(n^2)$.

Next, we compare the average case cost of *BBS* to the average case cost of *MINI*. Since *MINI* evaluates the skyline relative to a removed skyline point, the cost savings is dependant on the number of elements in the new skyline that were previously dominated by the removed skyline point. If this number of skyline points is s' and the total number of skyline points is s, then the cost of *MINI* relative to that of *BBS* is s'/s.

Therefore, the qualitative cost of using *LookOut* is less than that of iteratively running the *BBS* algorithm for continuous skyline computation.

4. Choice of indexing structure

In this section, we examine how the choice of the index can impact the performance of both static and continuous time-interval skyline performance. This section examines some of the differences between the ubiquitous R^*-tree which has been used for a number of the previously proposed skyline algorithms [24,14], and the quadtree, which is more efficient for computing skylines. The quadtree has the following two advantages over the R^*-tree for evaluating continuous time-interval skylines: (1) insertion into a quadtree is faster than into a R^*-tree, and (2) The quadtree-based traversal reduces the maximum number of heap elements during the best-first search. (The second advantage also applies to skyline computation over static datasets.)

The rationale behind the first point involves the complex node split operation of the R^*-tree that involves various sorting and grouping operations on index entries. In contrast, the split operation of the quadtree is much simpler, and merely divides the node in each dimension in half. For point data, such as that managed in skyline queries, the superior performance of the quadtree on inserts and updates has been noted in a study of a commercial DBMS [15]. This study of Oracle Spatial shows that quadtrees are significantly faster for index creation and updates of point data.

The intuition driving the second point above is as follows: first, each time the R^*-tree splits, its children are likely to overlap. No dominance relationship can be established between two overlapping leaf or non-leaf nodes, so neither will be able to prune the other from future consideration. Hence, both will be inserted into the heap. Contrast this with the node split of the quadtree, where no overlap exists, and at least one child is automatically dominated (and pruned) each time a split is performed.[1] Second, nodes in the quadtree will produce quite different distances from the origin for their internal data. This is because each quad occupies a region of space derived only from the structure of the quadtree, and not from the data as in the case of the R^*-tree. This means that the children of one quad will be fully expanded and mostly removed from the heap before the data contained in neighboring leaf and non-leaf nodes is entered into the heap.

To understand the heap size reduction benefit of quadtrees, consider the example shown in Fig. 6a. Nodes *A*, *B*, and *C* are inserted into the heap when their parent node is expanded. *D* is not inserted, because it is

[1] A question that a reader may ask is why not consider an R^+-tree instead of a quadtree. While a full exploration of this issue is beyond the scope of this paper, the quick answer is that the R^+-tree does not guarantee the pruning property of the quadtree, which is critical to the efficiency for skyline computation. The R^+-tree only addresses the non-overlapping problem of the R^*-tree, but at the expense of lower page occupancy.

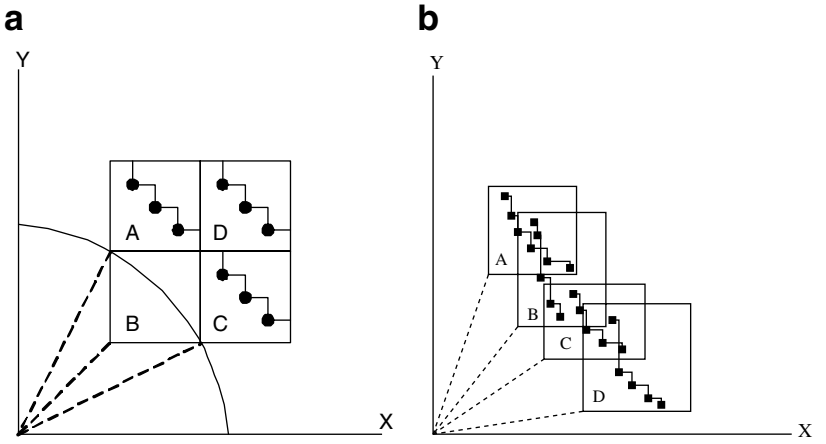


Fig. 6. Quadtree (a) and R^*-tree (b) nodes with local skylines. Distances to each represented by dashed lines.

automatically dominated by B. B is the first node popped from the heap, and its local skyline points are inserted back into the heap and ordered by the distance function. The distance that A and C are from the origin is represented by the quarter circle. Note that most of the area of B lies within this quarter circle. Any entries in B that lie within this circle are processed and removed from the heap before either A or C is expanded, thus resulting in a smaller heap. Contrast this to the worst case performance of the R^*-tree, seen in Fig. 6b. A, B, C, and D are added to the heap with similar distances. Hence, each is expanded in sequence before any of their individual data elements are processed.

5. Experimental evaluation

In this section, we present experimental results comparing *LookOut* with *BBS*, the best known method for computing skylines. We compre the quadtree with the R^*-tree [2], a variant of the R-tree. We first present results showing that for static skylines, using the quadtree significantly improves the performance over using an R^*-tree. We also show that the heap size is smaller when using the quadtree compared to the R^*-tree, implying that a smaller amount of memory is needed for computing skylines with the quadtree approach. (A low memory consumption is critical in streaming environments in which the system is evaluating multiple skylines concurrently.) We then present results for *LookOut* with the time-interval continuous skyline model.

5.1. Experimental study goal

In this study, our goal is to compare the performance of the R^*-tree and the quadtree for skyline query evaluation. The R^*-tree is chosen because indexed skyline query algorithms discussed previously have focused exclusively on the R-tree family of indices. Quadtrees have been shown to manage point data more effectively than the R-tree family in several notable experimental studies [15,13]. Since skyline queries deal exclusively with point data, it is for this reason we have chosen the quadtree as the best alternative. For a broader comparison beyond the scope of skyline queries for indices in the R-tree family and the quadtree, the interested reader may consult [15,13].

5.2. Data sets and experimental setup

The choice of datasets for experimental evaluation is always a challenging task. While the use of real datasets is preferable, a few selected real datasets do not necessarily bring out the effect of a range of data distributions. Luckily for skyline methods, it has been recognized that there are three critical types of data distributions that stress the effectiveness of skyline methods [4]. These three distributions are independent,

correlated, and anti-correlated. The correlated dataset is considered the easiest case for skyline computation since a single point close to the origin can quickly be used to prune all but a small portion of the data from consideration. The anti-correlated dataset is considered the most challenging of the three for skyline computation. This is because points in the skyline dominate only a small portion of the entire dataset. Larger numbers of skyline points exist for anti-correlated data for a given cardinality relative to either the independent or correlated cases.

To begin the discussion, first consider the different types of data distributions and the varying effects that these distributions have on the cost of computing the skyline operation. The two-dimensional case for each of the common data distributions that have been extensively considered in previous work are shown in Figs. 7–9. Only a small portion of the data (and hence only a small part of the data in the index) will be considered during the skyline evaluation of the correlated and independent cases, since each has a data point or points near

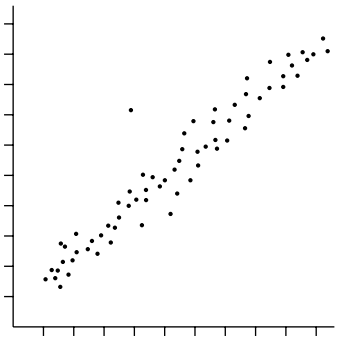


Fig. 7. Two-dimensional example of correlated data.

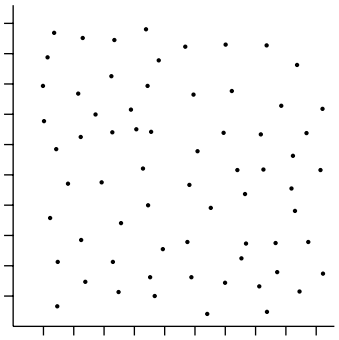


Fig. 8. Two-dimensional example of independent data.

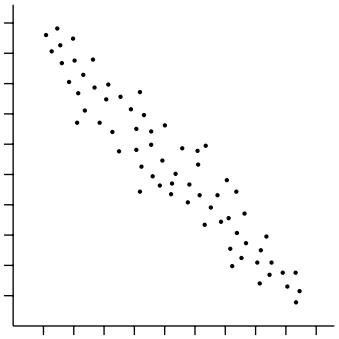


Fig. 9. Two-dimensional example of anti-correlated data.

the origin for sufficiently large cardinality values. These points will dominate all or most of the remaining points in the dataset, quickly pruning away the majority of the data from skyline consideration. The anti-correlated dataset is more challenging for skyline algorithms because it produces more skyline points for a given dataset cardinality (on average) than the other distributions. Hence, a greater number of points are considered for inclusion in the skyline, which means that more leaf-level nodes and inner nodes of a spatial index must be traversed by a skyline algorithm. While real datasets may have distributions that differ from these benchmarks, these three distributions present a wide and diverse range of distributions to test the performance of skyline algorithms.

Following well-established methodology set by previous research on skyline algorithms [17,24], we choose to use these three data distributions. We also test our methods on a variety of other data set parameters such as data cardinality and dimensionality. For generating these synthetic datasets, we use the skyline generator generously provided by the authors of [4]; using this, we created a number of datasets varying in cardinalities from 100 K to 5 M in two dimensions, for the three distributions already mentioned. We also created datasets varying the dimensionality between 2 and 5 while holding the cardinality fixed at 1 M entries. We test with these dimensionalities because they have been commonly tested elsewhere for indexed skyline operations [24,34].

Our experimental platform is built on top of the SHORE storage manager [5], which provides support for R^*-trees. We also implemented a quadtree indexing method in SHORE. Our quadtree implementation uses a simple mapping of the quadtree nodes to disk pages. Each leaf level quadtree node is one page in size. Non-leaf nodes are simply implemented as SHORE objects, that are packed into pages in the order of creation.

We implemented both *BBS* and *LookOut* on top of the SHORE R^*-tree and our quadtree index implementation in SHORE. To maintain consistency with the previous approach by Papadias et al. [24], and for ease of comparison, we set the R^*-tree node size at 4 KB for both leaf and non-leaf nodes. This results in R^*-tree leaf node capacities of between 330 and 165 data entries for dimensions 2 and 5, respectively. The linear splitting algorithm is choosen for the nodes of the R^*-tree. The non-leaf node capacities vary between 110 for two dimensions and 66 for five dimensions. We also used a 4 KB page size for our quadtree implementation, resulting in leaf node capacities that varied between 424 for two dimensions to 131 in five dimensions. The leaf node utilization for the R^*-tree is 71 percent for two-dimensional data for both the independent and anti-correlated datasets and 74 and 73 percent for five-dimensional data for the independent and anti-correlated datasets, respectively. The leaf node utilization for the quadtree is 61 and 53 percent for five-dimensional data for the independent and anti-correlated datasets, respectively, and 30 and 13 percent for five-dimensional data for the independent and anti-correlated datasets, respectively. We use a buffer pool size of 128 MB. For the *BBS* implementation, we followed the algorithm described in [24], and added the local skyline optimizations described in [25].

All experiments were run on a machine with a 1.7 GHz Intel Xeon processor, with 512 MB of memory and a 40 GB Fujitsu SCSI hard drive, running Debian Linux 2.6.0.

5.3. R^*-tree v/s quadtree for skyline computation

In this section, we examine the effect of the underlying index structure on the performance of *static* skyline computation. In other words, we show the effect of the choice of index on the *BBS* algorithm [24,25]. We focus on two different properties that affect skyline query performance: the dataset dimensionality and cardinality. This methodology is consistent with the performance study of [24]. In the interest of space, we only present results for the anti-correlated and independent cases, which is also consistent with previous studies [24,25].

We measure the number of page accesses in our experiments instead of the disk access cost (DAC) [20] because the DAC is a measure of the number of all nodes of a tree that are read during a query. For members of the R-tree family, this closely matches the number of page accesses, since R-tree nodes are mapped directly to pages. For inner nodes of a packed quadtree, this is not the case since many inner nodes can be mapped to one single page. Hence, we measure page accesses as a more fair comparison for the work done by both data structures.

5.3.1. Effect of cardinality

In this experiment, we explore the effect of the data cardinality. Following the approach in [24], we fix the dimensionality at 2, and vary the cardinality between 100 K and 5 M. As in [24], we report three different graphs for each experimental setting: the CPU time versus cardinality, the maximum size of the heap versus cardinality, and the number of page accesses versus cardinality. The results for this experiment are shown in Figs. 10–15.

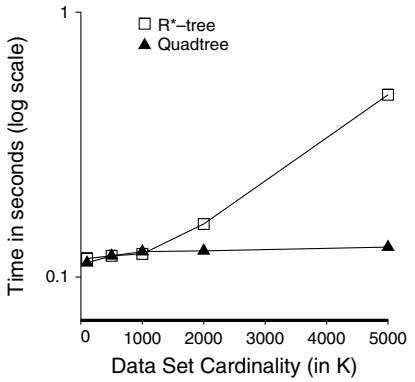


Fig. 10. Execution time: indep. dist., 2 dim., varying cardinality.

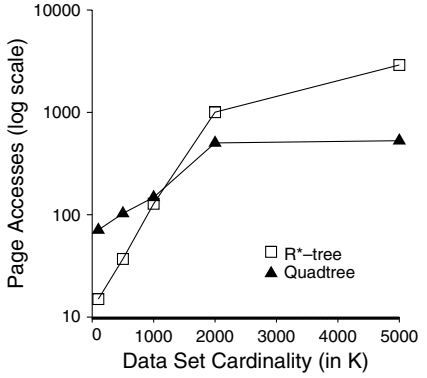


Fig. 11. Page accesses: indep. dist., 2 dim., varying cardinality.

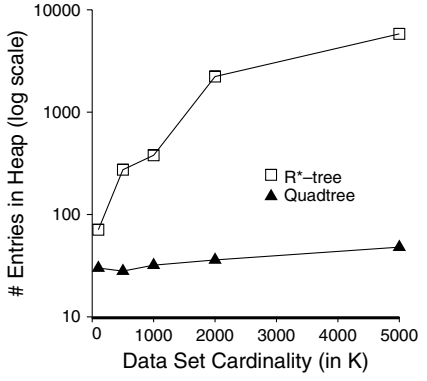


Fig. 12. Maximum heap size: indep. dist., 2 dim., varying cardinality.

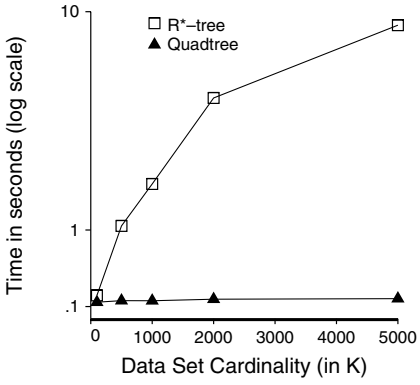


Fig. 13. Execution time: anti-corr. dist., 2 dim., varying cardinality.

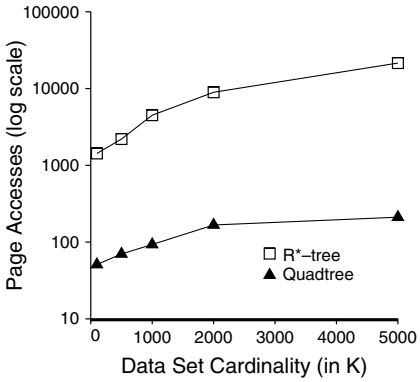


Fig. 14. Page accesses: anti-corr. dist., 2 dim., varying cardinality.

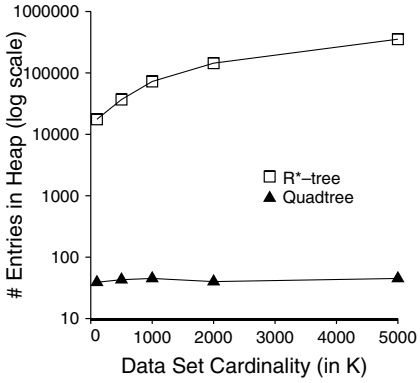


Fig. 15. Maximum heap size: anti-corr. dist., 2 dim., varying cardinality.

Figs. 10 and 13 present the execution times for varying cardinality. In the independent case (Fig. 10), both the R^*-tree and quadtree based methods are comparable until the dataset size is over one million entries; for the larger data sizes, the quadtree approach is significantly faster. For the anti-correlated dataset (Fig. 13) the quadtree approach is significantly faster, and its relative performance improves as the data cardinality increases – for the 5000 K data it is two orders of magnitude faster than the R^*-tree approach. This difference occurs because of the lower page accesses for the quadtree and smaller maximum heap size.

In Fig. 11, we notice that the number of page accesses for the independent case for the R^*-tree is 2–4 times that of the quadtree for the 2 M and 5 M data file sizes. For these two file sizes, the quadtree outperforms the R^*-tree by significant amounts (see Fig. 10).

From Fig. 14, we observe that the R^*-tree performs about an order of magnitude more page accesses than the quadtree. These increased page accesses are attributable to the better pruning techniques of the quadtree caused by the node overlaps of the R^*-tree (as discussed in Section 4). The quadtree accesses fewer leaf and non-leaf nodes than the R^*-tree because it can prune away more nodes that are dominated by the discovered skyline points. As a side note, for the 100 K data size the quadtree approach actually performs a few more reads (31 versus 21), which is attributable to a larger tree height for the quadtree (5 versus 3) relative to the R^*-tree index, and the relatively simple packing of quadtree nodes in our implementation.

The maximum heap sizes in Figs. 12 and 15 show a large improvement for the quadtree method for all file sizes, since the quadtree is accessing fewer leaf and non-leaf nodes than the R^*-tree due to its non-overlapping space partition. In addition, the nodes that it does access are processed much more serially than the R^*-tree, whose overlapping leaf and non-leaf nodes are expanded into the heap at similar times because they have similar distances from the origin. This results in the fewer page accesses and the smaller maximum heap size for the quadtree (see Section 4 for the detailed analysis).

5.3.2. The effect of dimensionality

In this experiment, we examine the effect of data dimensionality. As in [24], we fix the data cardinality at 1 million tuples, and vary the dimensionality from 2 to 5. The results for this experiment are shown in Figs. 16–21.

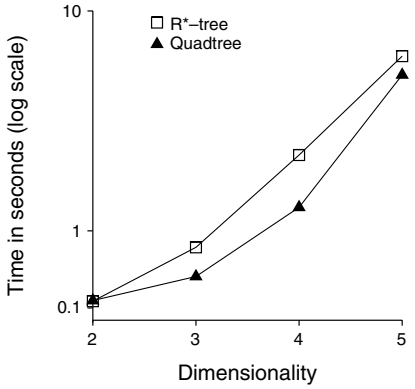


Fig. 16. Execution time: indep. dist., varying dim., 1 M cardinality.

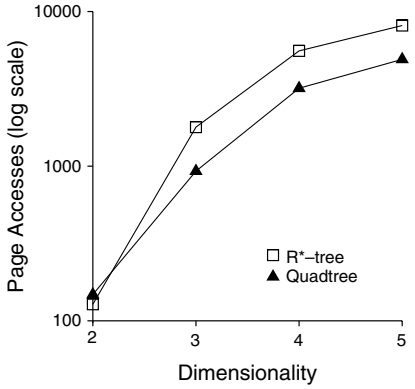


Fig. 17. Page accesses: indep. dist., varying dim., 1 M cardinality.

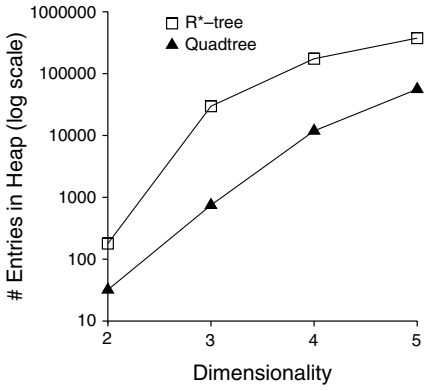


Fig. 18. Maximum heap size: indep. dist., varying dim., 1 M cardinality.

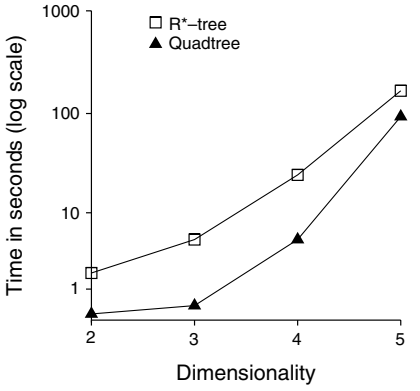


Fig. 19. Execution time: anti-corr. dist., varying dim., 1 M cardinality.

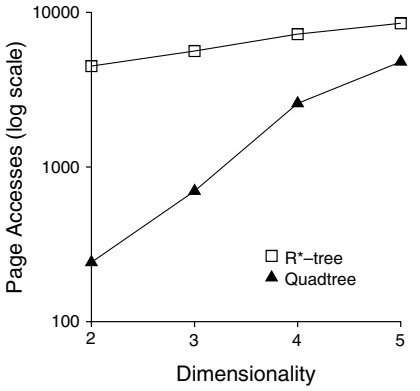


Fig. 20. Page accesses: anti-corr. dist., varying dim., 1 M cardinality.

The execution time graphs for increasing dimensionality are seen in Fig. 16 for the independent and Fig. 19 for the anti-correlated datasets. For the independent case (Fig. 16), the quadtree is about 2–4 times faster when dimensionality is higher than two. For the anti-correlated dataset (Fig. 19), the quadtree is more than an order of magnitude faster than the R^*-tree when the dimensionality is lower than five. These benefits are because the quadtree approach incurs significantly fewer pages accesses and has fewer entries in the heap.

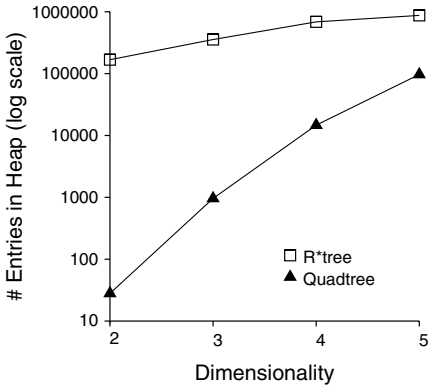


Fig. 21. Maximum heap size: anti-corr. dist., varying dim., 1 M cardinality.

In Fig. 17, the number of page accesses for the independent case for the R^*-tree is similar for two dimensions. As seen in Section 5.3.1, the quadtree and R^*-tree had comparable performance for two dimensions when the cardinality was less than 2 M. For higher dimensionality, the quadtree obtains cost savings of about 2–3 times. This is attributable to the higher chance for dead space and overlap amongst R^*-tree nodes as dimensionality increases, which exposes the relative superior pruning of the quadtree, leading to more page accesses and heap accesses for the R^*-tree (as discussed in Section 4).

In Fig. 20, the R^*-tree performs about an order of magnitude more page accesses than the quadtree in two and three dimensions, about three times more page accesses in four dimensions, and about twice as many page accesses in five dimensions. There are two competing factors at work here. First, the superior pruning of the quadtree results in a lower number of page accesses, relative to the R^*-tree. Second, the increasing dimensionality means that more skyline points exist for the anti-correlated dataset and the quadtree has to access more data pages to find them all. The R^*-tree accesses slightly more pages as well (about twice as many in five dimensions as in two), but the fact that it was already accessing so many in two dimensions means that the increase in the rate of page access with dimensionality is not as remarkable as that of the quadtree.

The maximum heap size in Fig. 18 shows a savings of about an order of magnitude for the quadtree over the R^*-tree. This is again attributable to the pruning techniques of the quadtree, as discussed in Section 5.3.1.

Fig. 21 shows a similar trend for heap size as dimensionality increases as was witnessed for the number of page accesses. The same two competing factors are causing this. First, the superior pruning of the quadtree gives rise to a smaller maximum heap size. Second, the increasing rate of page accesses with dimensionality means more pages will have similar distances as the dataset fans out. Thus, more pages will be expanded and insert their entries into the heap at about the same time (see Section 4 for details).

5.3.3. Summary

In summary the quadtree index is a much more efficient indexing structure than the R^*-tree for computing skylines. The benefits of using a quadtree are generally higher for larger datasets, and for lower dimensionality. In many cases, the quadtree approach is more than an order of magnitude faster than the R^*-tree approach. The benefits are the most significant for the anti-correlated dataset. In addition to being fast, the quadtree approach also results in a significantly smaller maximum heap size.

5.4. The continuous time-interval skyline

In this section, we examine the performance of *LookOut* relative to a naive method of executing the *BBS* algorithm to compute the skyline whenever anything changes. This method is referred to as *CBBS*, and can be considered the best alternative method for computing continuous skylines.

For this experiment, the data structures are entirely memory resident, to mimic the application of continuous skyline in streaming applications where such main memory assumptions are common and often the

preferred environment (for example [17] also assumes that there is enough main memory). In the naïve *CBBS* case, a binary heap ordered on data point expiration time is maintained, so that when a point expires, it can be deleted and the *BBS* skyline algorithm run to reevaluate the skyline. Whenever a data entry arrives, it is inserted into both the heap and the R^*-tree, and the skyline is reevaluated by rerunning *BBS*.

As before, we used synthetic datasets and vary both the cardinality and the dimensionality. For the dimensionality tests, we vary d between 2 and 5 and fix the cardinality at 10 K entries. For the cardinality test, we fix d at 2 and vary the dataset cardinality from 10 K to 50 K.

Two different techniques are used to assign data points an arrival and an expiration time, and results for both techniques are presented. For the first technique, we randomly pick an arrival time between 0 and 100 K. Then, we pick the departure time randomly between the arrival time and 100 K. For the second technique, data points are again assigned an arrival time between 0 and 100 K, but the expiration time is chosen randomly between 1 and 10 percent of the total time interval, i.e. between 1000 and 10,000 time points later than the arrival time. The data generated using the first technique is used to evaluate the performance of *LookOut* when the time interval varies widely; the second data generation technique is used to evaluate the effect on performance when the time intervals have a constrained size.

The results that we present are generated by running each dataset from time 0 to 100,001. During this time, each data point will arrive and be deleted following its expiration. The skyline is continuously updated over the course of the 0–100,000 time interval. We present results indicating the throughput in *elements per second (eps)* that can be achieved by *LookOut*. Note that this metric reflects the time to insert or delete an element as it arrives or expires, plus the time to update the continuous skyline. For each experiment, we consider the implementation of *CBBS* and *LookOut* using the R^*-tree and the quadtree. For *CBBS* we use the label *CBBS-R* and *CBBS-Q* for the R^*-tree and the quadtree index implementations, respectively. Similarly *LookOut-R* and *LookOut-Q* refer to the R^*-tree and quadtree implementation of *LookOut*, respectively. The y-axis in all figures uses a log scale to show the workload execution time.

5.4.1. Cardinality

In this test, we vary the data cardinality from 10 K to 50 K. The results of this experiment using data generated with the first technique (expiration times randomly chosen between the arrival time and 100 K) are shown in Figs. 22–24, for the three data distributions. In these figures, we observe that the execution time for *LookOut* with a quadtree relative to *CBBS* is more than two orders of magnitude better. *LookOut* with the R^*-tree is between 2 and 6 times faster than *CBBS* in the anticorrelated case and almost twice as fast in the independent case for dataset cardinalities greater than 20 K. In the correlated case, *LookOut* with the R^*-tree achieves only a small improvement over *CBBS* with an R^*-tree. The superior performance of *LookOut* with respect to *CBBS* irrespective of the underlying data structure is expected due to the efficiencies of *LookOut* in updating the skyline with each new insertion or deletion instead of recomputing it from scratch as *CBBS* does. There is a more marked improvement in *LookOut* relative to the *CBBS* algorithm with the

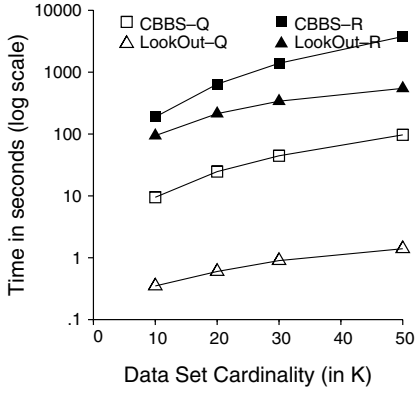


Fig. 22. Execution time: continuous, random time interval length, anti-corr., 2 dim., varying cardinality.

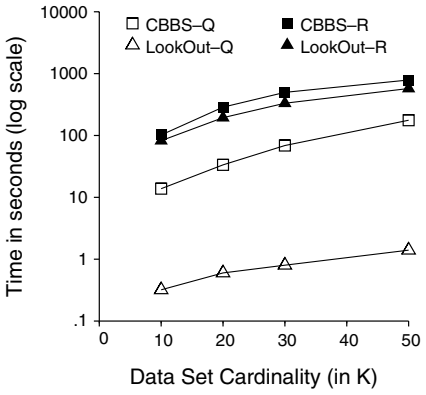


Fig. 23. Execution time: continuous, random time interval length, indep., 2 dim., varying cardinality.

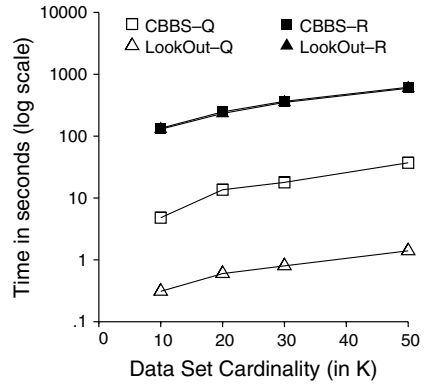


Fig. 24. Execution time: continuous, random time interval length, corr., 2 dim., varying cardinality.

quadtree than with the R^*-tree because of the improvements of the quadtree over the R^*-tree for skyline evaluation already mentioned and because the insertions and deletions with the quadtree are very fast. The overhead of the R^*-tree limits the amount of performance improvement that *LookOut* can achieve.

The results of the experiment using data generated with the second technique (expiration times randomly chosen between 1 and 10 percent of the total time interval) are shown in Figs. 25–27, for the three data distributions. The trends in the data are similar, with *LookOut* with the quadtree again outperforming *LookOut*

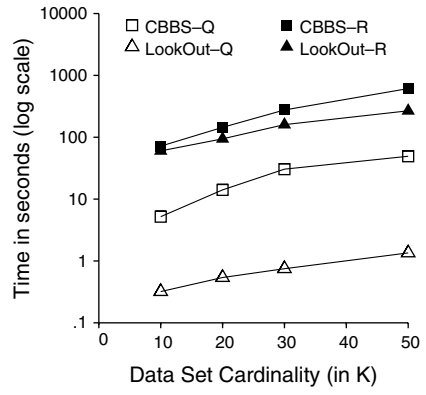


Fig. 25. Execution time: continuous, 1–10 percent time interval length, anti-corr., 2 dim., varying cardinality.

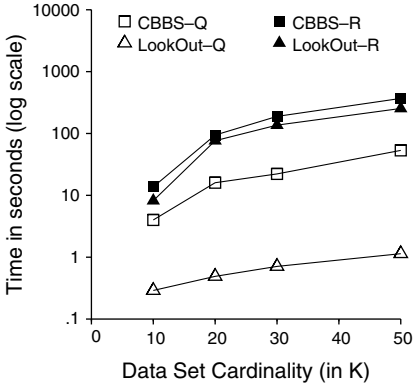


Fig. 26. Execution time: continuous, 1–10 percent time interval length, indep., 2 dim., varying cardinality.

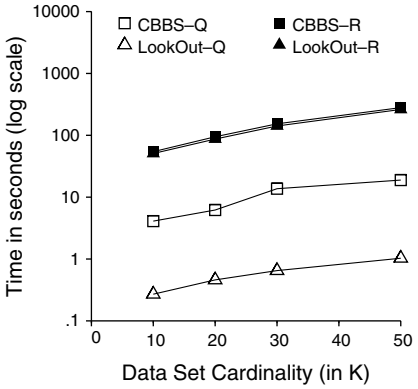


Fig. 27. Execution time: continuous, 1–10 percent time interval length, corr., 2 dim., varying cardinality.

with the R^*-tree and *CBBS*, regardless of indexing structure by at least an order of magnitude. *LookOut-R* also outperforms *CBBS-R* by a factor of 2–3 in the anti-correlated case.

Table 3 presents data on the maximum and average processing delays for *LookOut* for this experiment. These results indicate that *LookOut* can process about 45,248 eps for the anti-correlated dataset, and about 46,728 eps for the independent dataset. (Note 1000/0.0221 = 45,248.)

5.4.2. Dimensionality

The results for the dataset dimensionality tests using data generated with the first technique (expiration times randomly chosen between the arrival and 100 K) are presented in Figs. 28–30. We observe that the execution time for *LookOut* with an R^*-tree relative to *CBBS* with an R^*-tree is about twice as fast for the independent case for each dimensionality, and between 2 and 9 times better for the anti-correlated case, depending

Table 3
Delays in processing inserts and deletes for *LookOut*, varying cardinality, with 2 dim. data

Cardinality in K	Maximum anti-correlated delay	Maximum independent delay	Average anti-correlated delay	Average independent delay
10	1.21	1.55	0.0291	0.0273
20	1.42	1.72	0.0250	0.0235
30	3.31	2.99	0.0236	0.0223
50	3.41	4.57	0.0221	0.0214

Delays in ms.

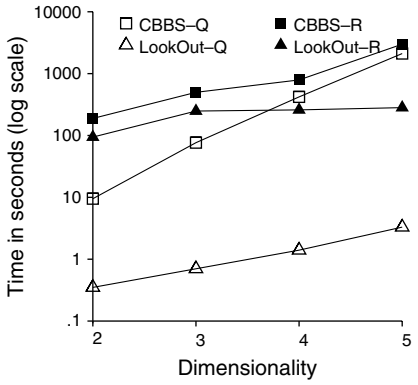


Fig. 28. Execution time: continuous, random time interval length, anti-corr., varying dim., 10 K cardinality.

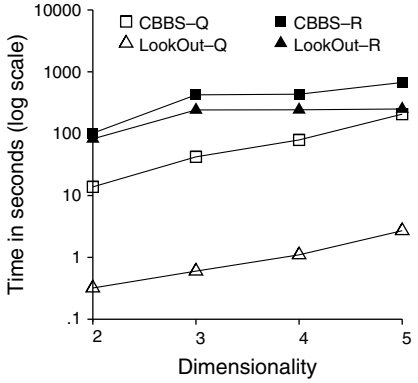


Fig. 29. Execution time: continuous, random time interval length, indep., varying dim., 10 K cardinality.

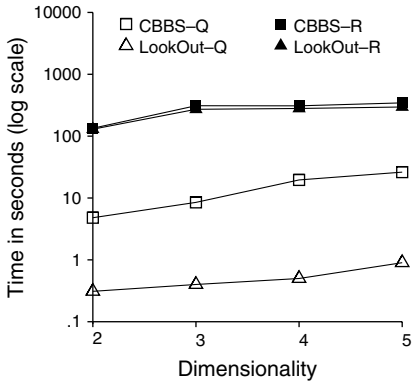


Fig. 30. Execution time: continuous, random time interval length, corr., varying dim., 10 K cardinality.

on the dimensionality. The best algorithm is again *LookOut* with the quadtree, as it only incrementally recomputes the skyline on inserts and deletes, and uses the faster inserts and deletes methods of the quadtree. It is an order of magnitude better than *CBBS* with an underlying quadtree for all data distributions for all dimensionalities and is more than two orders of magnitude better in high dimensionality for the anticorrelated case. The results of the experiment using data generated with the second technique (expiration times randomly chosen between 1 and 10 percent of the total time interval) are shown in Figs. 31–33.

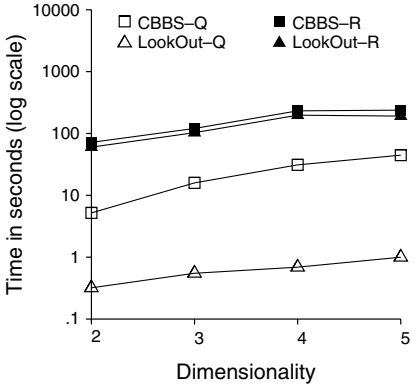


Fig. 31. Execution time: continuous, 1–10 percent time interval length, anti-corr., varying dim., 10 K cardinality.

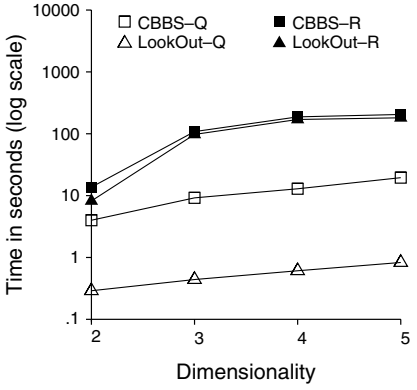


Fig. 32. Execution time: continuous, 1–10 percent time interval length, indep., varying dim., 10 K cardinality.

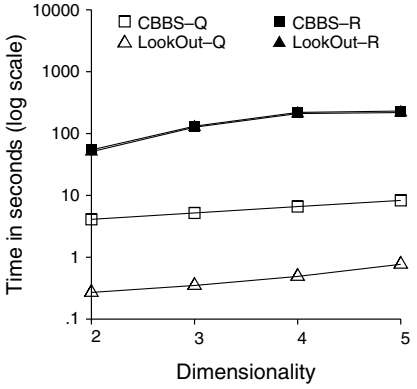


Fig. 33. Execution time: continuous, 1–10 percent time interval length, corr., varying dim., 10 K cardinality.

From these figures we observe that the execution time for *LookOut* is less than *CBBS* with each respective data structure. With the quadtree, *LookOut* is more than an order of magnitude faster than *CBBS* with the quadtree. With the R^*-tree, *LookOut* is faster on average by a factor of 2–3. In the anti-correlated case, the rate of increase for the *CBBS* algorithm is higher as d increases than it is for *LookOut*, indicating that *LookOut* scales better for increasing d.

Table 4
Delays in processing inserts and deletes for *LookOut*, for varying dim., 10 K cardinality

Dimensionality	Maximum anti-correlated delay	Maximum independent delay	Average anti-correlated delay	Average independent delay
2	1.21	1.55	0.0291	0.0273
3	1.54	5.65	0.0613	0.0499
4	3.26	9.40	0.1297	0.0959
5	4.45	14.80	0.3390	0.2598

Delays in ms.

In Table 4, we present the maximum and average processing delays for the *LookOut* algorithm. These results indicate that *LookOut* can support a throughput rate of about 36,630 eps and 34,364 eps for the independent and anti-correlated cases, respectively, at a dimensionality of 2. For dimensionality 5, the throughput rates are about 3849 eps and 2950 eps for the independent and anti-correlated cases.

5.5. Comparison with the eager and lazy techniques

In this section, we compare the performance of *LookOut* with that of the *eager* and *lazy* techniques of [34]. The code for these techniques was obtained from the first author's website and compiled for Linux. Following the approach of [34], we experiment with dimensions 2, 3, and 4, windows of size 200, 400, 800, and 1600 tuples, and report back the per tuple processing times in milliseconds. The independent and anti-correlated datasets were generated with the data set generator also provided from the first author's website. Each data set was modified for *LookOut* by assigning each tuple an expiration time equal to the arrival time plus a number of time units equal to the size of the window. This means that both *LookOut* and *lazy* and *eager* have the exact same data points available for inclusion in the skyline at any one time and produce the same skyline results. The results for varying dimensionality with a fixed 800 tuple window size are presented in Fig. 34a for independent data and in Fig. 34b for anti-correlated data. The results for three-dimensionals with a varying tuple window size are presented in Fig. 35a for independent data and in Fig. 35b for anti-correlated data.

In Fig. 34a and b, *LookOut* using the quadtree performs better than the *eager* technique for the independent dataset and about an order of magnitude better than either *eager* or *lazy* for the anti-correlated dataset for dimensionality 3 and 4, while handling the more general expiration time model. It achieves similar results for all window sizes in Fig. 35a and b. The performance advantage is largely due to the better update performance of the quadtree in these experiments, since *LookOut* with the R^*-tree was much slower, particularly for the independent dataset. *LookOut* does not perform as well as the *lazy* technique for the independent dataset. This is because the size of the skyline is much smaller than in the anti-correlated case, so the benefits of using

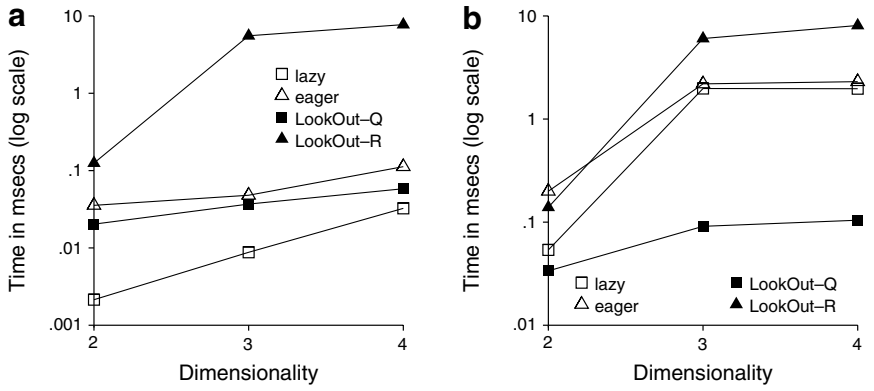


Fig. 34. The per tuple processing costs for varying dimensionality and a fixed window of size 800 for (a) independent and (b) anti-correlated data.

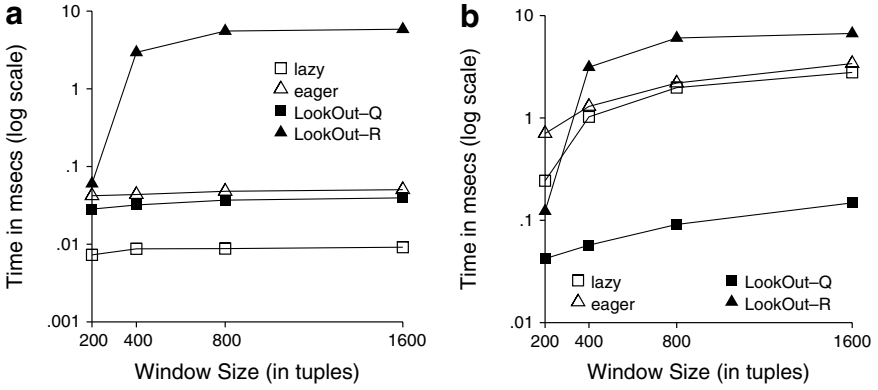


Fig. 35. The per tuple processing costs for three-dimensional data and a varying window size for (a) independent and (b) anti-correlated data.

the quadtree is much reduced. The important observation from these experiments is that the performance of *LookOut* is better than *eager* in all cases and better than *lazy* in the anti-correlated case even when handling the more restricted time model.

6. Conclusions and future work

In this paper, we have introduced the continuous time-interval skyline operation. This operation continuously evaluates a skyline over multidimensional data in which each element is valid for a particular time range. We have also presented *LookOut*, an algorithm for efficiently evaluating continuous time-interval skyline queries. Detailed experimental evaluation shows that this new algorithm is usually more than an order of magnitude faster than existing methods for continuous skyline evaluation. We leave as future work the possibility of saving computations involved in finding the skylines for differing combinations of attribute sets that may share certain subsets of attribute combinations.

We have also exposed several inherent problems with using the R^*-tree index for evaluating a skyline. The primary reason for the inefficiency of the R^*-tree for skyline computation is the overlap of the bounding box keys, which results in poor subtree pruning of the index non-leaf and leaf nodes that are examined during the skyline computation. We have shown that the quadtree index is a much more efficient index structure for evaluating skylines. The non-overlapping partitioning characteristics of the quadtree leads to a natural decomposition of space that can more effectively prune the index nodes that must be searched. An extensive experimental evaluation shows that using a quadtree can result in a continuous skyline evaluation method that can achieve high throughput and can also dramatically speed up traditional static skyline computation.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0414510. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] www.orbitz.com.
[2] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1990, pp. 322–331.
[3] C. Böhm, H. Kriegel, Determining the convex hull in large multidimensional databases, in: Data Warehousing and Knowledge Discovery (DaWaK), 2001, pp. 294–306.
[4] S. Borzsonyi, D. Kossmann, K. Stocker, The skyline operator, in: Proceedings of International Conference on Data Engineering (ICDE), 2001, pp. 421–430.

[5] M. Carey, D. DeWitt, M. Franklin, et al., Shoring up persistent applications, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1994, pp. 383–394.
[6] M. Carey, D. Kossmann, On saying enough already! in SQL, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1997, pp. 219–230.
[7] J. Chomicki, P. Godfrey, J. Gryz, D. Liang, Skyline with presorting, in: Proceedings of International Conference on Data Engineering (ICDE), 2003, pp. 717–720.
[8] I. Gargantini, An effective way to represent quadtrees, Communications of the ACM 25 (12) (1982) 905–910.
[9] A. Guttman, *R*-tree: a dynamic index structure for spatial searching, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
[10] G. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Transactions on Database Systems (TODS) 24 (2) (1999) 265–318.
[11] G.R. Hjaltason, H. Samet, Speeding up construction of PMR quadtree-based spatial indexes, The VLDB Journal 11 (2) (2002) 109–137.
[12] Z. Huang, H. Lu, B. Ooi, A. Tung, Continuous skyline queries for moving objects, Transactions on Knowledge and Data Engineering (TKDE) 18 (12) (2006) 1645–1658.
[13] Y.J. Kim, J.M. Patel, Rethinking choices for multidimensional point indexing: making the case for the often ignored quadtree, 2007.
[14] D. Kossmann, F. Ramsak, S. Rost, Shooting stars in the sky: an online algorithm for skyline queries, in: Very Large Databases (VLDB), 2002, pp. 275–286.
[15] R.K.V. Kothuri, S. Ravada, D. Abugov, Quadtree and *R*-tree indexes in oracle spatial: a comparison using GIS data, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2002, pp. 546–557.
[16] H. Kung, F. Luccio, F. Preparata, On finding the maxima of a set of vectors, Journal of the ACM 22 (4) (1975) 469–476.
[17] X. Lin, Y. Yuan, W. Wang, H. Lu, Stabbing the sky: efficient skyline computation over sliding windows, in: Proceedings of International Conference on Data Engineering (ICDE), 2005, pp. 502–513.
[18] D. Littau, D. Boley, Streaming data reduction using low-memory factored representations, Information Sciences 176 (14) (2006) 2016–2041.
[19] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, Y. Theodoridis, *R*-Trees: Theory and Applications, Series in Advanced Information and Knowledge Processing, Springer, 2005.
[20] Y. Manolopoulos, Y. Theodoridis, V. Tsotras, Advanced Database Indexing, Springer, 1999.
[21] J. Matoušek, Computing dominances in E^n, Information Processing Letters 38 (5) (1991) 277–278.
[22] D. McLain, Drawing contours from arbitrary data points, The Computer Journal 17 (4) (1974) 318–324.
[23] M. Morse, J.M. Patel, W.I. Grosky, Efficient continuous skyline computation, in: Proceedings of International Conference on Data Engineering (ICDE), 2006, pp. 108.
[24] D. Papadias, Y. Tao, G. Fu, B. Seeger, An optimal and progressive algorithm for skyline queries, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2003, pp. 467–478.
[25] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Transactions on Database Systems (TODS) 30 (1) (2005) 41–82, March.
[26] C. Papadimitriou, M. Yannakakis, Multiobjective query optimization, in: Proceedings of ACM Principles of Database Systems, 2001, pp. 52–59.
[27] F. Preparata, M. Shamos, Computational Geometry – An Introduction, Springer, 1985.
[28] N. Roussopoulos, S. Kelly, F. Vincent, Nearest neighbor queries, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1995, pp. 71–79.
[29] H. Samet, The quadtree and related hierarchical data structures, Computing Surveys 16 (4) (1984) 187–260.
[30] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufman, 2006.
[31] R. Steuer, Multiple Criteria Optimization, Wiley, NY, 1986.
[32] I. Stojmenovic, M. Miyakawa, An optimal parallel algorithm for solving the maximal elements problem in a plane, Parallel Computing 7 (2) (1988) 249–251.
[33] K. Tan, P. Eng, B. Ooi, Efficient progressive skyline computation, in: Very Large Databases (VLDB), 2001, pp. 301–310.
[34] Y. Tao, D. Papadias, Maintaining sliding window skylines on data streams, Transactions on Knowledge and Data Engineering (TKDE) 18 (2) (2006) 377–391.
[35] W.-T. Wong, F.Y. Shih, T.-F. Su, Thinning algorithms based on quadtree and octree representations, Information Sciences 176 (10) (2006) 1379–1394.